NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THREE-DIMENSIONAL PATH PLANNING FOR THE NPS II AUV

by

Tymothy Wayne Caddell

December 1991

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited.

T257747

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
THREE-DIMENSIONAL PATH PLANNING FOR THE NPS II AUV (U)

12. PERSONAL AUTHOR(S)
Caddell, Tymothy Wayne

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 10/89 TO 12/91	December 1991	181

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Path planning, Three-dimensional path planning, polyhedral path planning Autonomous Underwater Vehicle, NPS II AUV, obstacle representation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Naval Postgraduate School in Monterey, California is currently working on an ongoing project for research in autonomous underwater vehicle (AUV) technology. This pro ject comprises two areas of research. The first area is research conducted on the system, NPS II AUV. The second area is a computer simulation of the actual system. One topic which is vital to both areas is three-dimensional path planning. The concept of three-dimensional path planning is on the order of magnitude of polynomial time and current research in this area is limited. This paper reviews my findings and submits an algorithm which finds a best path in a three-dimensional environment, while avoiding all known polyhedral obstacles. The algorithm's concept is to reduce the three-dimensional world to a series of two-dimensional representations, allowing the algorithm to use tangential lines created from the start to nodes on the polygons lying between the start and goal, from nodes on polygons to other polygon nodes and finally, from polygon nodes to the goal.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Yutaka Kanayama	22b. TELEPHONE (Include Area Code): (408) 646-2095	22c. OFFICE SYMBOL: CS/Ka

Approved for public release; distribution is unlimited

THREE-DIMENSIONAL PATH PLANNING FOR THE NPS II AUV

by

Tymothy Wayne Caddell
Captain, United States Army
B.S., United States Military Academy, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December, 1991

ABSTRACT

The Naval Postgraduate School in Monterey, California is currently working on an ongoing project for research in autonomous underwater vehicle (AUV) technology. This project comprises two areas of research. The first area is research conducted on the system, NPS II AUV. The second area is a computer simulation of the actual system. One topic which is vital to both areas is three-dimensional path planning. The concept of three-dimensional path planning is on the order of magnitude of polynomial time and current research in this area is limited. This paper reviews my findings and submits an algorithm which finds a best path in a three-dimensional environment, while avoiding all known polyhedral obstacles. The algorithm's concept is to reduce the three-dimensional world to a series of two-dimensional representations, allowing the algorithm to use tangential lines created from the start to nodes on the polygons lying between the start and goal, from nodes on polygons to other polygon nodes and finally, from polygon nodes to the goal.

C154

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. AUV CONCEPT 4

C. PATH PLANNING AND REPLANNING 4

D. OBJECTIVES 5

E. THESIS ORGANIZATION 6

II. NPS AUV PROJECT AND RELATED WORK 7

A. SURVEY OF PREVIOUS WORKS 7

1. Autonomous Underwater Vehicle Projects . . . 7

2. Path Planning 10

B. THE NPS AUV PROJECT 16

1. The NPS AUV II Vehicle 16

2. Computer Simulation 18

III. OBSTACLE REPRESENTATION 20

IV. PATH PLANNING 23

A. BASIC CONCEPTS OF THREE DIMENSIONAL PATH PLANNING 23

1. Polyhedron Intersection 24

a. General Description. 24

b. Directed Ray. 25

c. Plane Equation. 26

d. Intersection Point. 27

e. Inside Polyhedron Check. 29
2. Plane Types 32
3. Two-Dimensional Polygons 32
4. Number and Types of Paths 34
5. Tangents 37
a. Plus and Minus Tangents 37
b. Vertical Tangents 44
6. Path Representation 46
7. Active Node List and Active Paths Heuristic 48
8. Visibility 50
B. THREE-DIMENSIONAL PATH PLANNING 54
1. Initial Paths 55
2. Extending Partial Paths 60
a. Search Technique 60
b. Extending the Shortest Path 61
c. Vertical Extension 62
(1) Horizontal Extension 66
C. FINDINGS AND EXAMPLES 67
1. One Polyhedron World 68
2. Two Polyhedron World 71
V. RECOMMENDATIONS AND CONCLUSIONS 75
LIST OF REFERENCES 78
APPENDIX A 80
APPENDIX B 85
APPENDIX C 89
APPENDIX D 96

APPENDIX E 98
APPENDIX F 101
APPENDIX G 121
APPENDIX H 126
APPENDIX I 131
APPENDIX J 135
APPENDIX K 145
APPENDIX L 148
APPENDIX M 154
INITIAL DISTRIBUTION LIST 169

LIST OF FIGURES

Figure 1 - NPS II AUV 17
Figure 2 - Polyhedron Link List 22
Figure 3 - 2D Polygon List 33
Figure 4 - Plus Tangents 38
Figure 5 - Minus Tangents 38
Figure 6 - Plus Tangents for Polyhedron World 38
Figure 7 - Minus Tangents for Polyhedron World 38
Figure 8 - Plane Polyhedron Intersection 39
Figure 9 - Direction Heuristic Check 41
Figure 10 - Tangents Among Polygons 43
Figure 11 - Vertical Tangent 45
Figure 12 - Vertical Points for One Polygon 46
Figure 13 - Vertical Nodes for More Than One Polygon . 46
Figure 14 - Partial Path List 47
Figure 15 - Active Node List 49
Figure 16 - Last Node Goal Visibility 51
Figure 17 - Points found for both Tangents 51
Figure 18 - Direction Heuristic Characteristics 53
Figure 19 - One World Polyhedron (Top Down View) . . . 69

LIST OF TABLES

Table I - TEXAS A&M MISSION REQUIREMENTS 8

Table II - TEXAS A&M VEHICLE CHARACTERISTICS 8

Table III - DARPA MISSION AND VEHICLE CHARACTERISTICS . 9

Table IV - PSEUDO LANGUAGE FOR INTERSECT POLYHEDRON . . 25

Table V - FIND PLANE EQUATION SOURCE CODE 27

Table VI - INTERSECTION POINT SOURCE CODE 29

Table VII - PSEUDO LANGUAGE FOR POLYHEDRON INTERSECTION 31

Table VIII - BUILD TWO-DIMENSIONAL POLYGON 34

Table IX - POSSIBLE PATHS FOR POLYHEDRON WORLD . . . 36

Table X - PLUS TANGENT PSEUDO LANGUAGE 40

Table XI - DETERMINING CLOSEST TANGENT POINT SOURCE CODE 42

Table XII - FINDING APPROPRIATE TANGENTS 44

Table XIII - CHECK ACTIVE NODE PSEUDO LANGUAGE 50

Table XIV - PSEUDO LANGUAGE FOR VISIBILITY 52

Table XV - PATH PLANNER PSEUDO LANGUAGE 55

Table XVI - PSEUDO LANGUAGE FOR START FINDING PATHS . . 56

Table XVII - PSEUDO LANGUAGE FOR EXTEND_PATH 62

Table XVIII - PSEUDO LANGUAGE FOR FUNCTION UP, OVER, UP 64

Table XIX - ONE POLYHEDRON WORLD 69

Table XX - SECOND POLYHEDRAL LISTING 72

Figure 20 - Initial Partial Paths 69
Figure 21 - Second Example for 1 Polyhedron 70
Figure 22 - 2 Polyhedral World and Initial Partial Paths 72
Figure 23 - Extending a Shortest Path in a Two Polyhedral World 72
Figure 24 - Initial Paths for Two Polyhedral World (Example 2) 73
Figure 25 - First Iteration 73
Figure 26 - Second Iteration 74
Figure 27 - Third Iteration 74
Figure 28 - Fourth Iteration 74

ACKNOWLEDGMENT

With not only the pressures of academia to include this thesis, I also had to contend with an extended hospital visit and three major surgeries during the last six months. Without my belief in the Lord Jesus Christ and His love for me, my experiences and outlook of my future would have been bleak. The patience, love, and support of my wife, Becky, during this time gave me the confidence and stamina to endure. And finally, but not last, I am forever greatful for the patience and guidance my advisor, Yutaka Kanayama, gave me. This academic adventure was significantly easier because of these two people.

I. INTRODUCTION

A. BACKGROUND

The last few years, technology has advanced significantly to see the use of unmanned vehicles substantially increase for land, sea and air applications. With regards to military applications, tacticians have used this technology to improve and extend the range and effectiveness of sensors and weapons. In the recent United States, and allied coalition forces' invasion of Iraq in DESERT STORM, both land and sea forces used remotely operated planes to perform reconnaissance of enemy forces with remarkable success. There are other examples of the successful use of remotely operated vehicles. For example, the Israeli Air Force successfully used remotely operated vehicles to perform reconnaissance and intelligence collection of enemy strong points prior to air campaigns conducted in 1982, resulting in zero losses for the Israelis and substantial damage inflicted on enemy forces. (Floyd 91).

Besides air operations, the concept of remotely operated vehicles can be incorporated in naval operations. The United States Navy's current maritime strategy calls for its forces to be forward deployed as possible. This results in

US Naval forces operating in or near enemy controlled waters. However, since Soviet strategies for both land and sea operations call for military operations to be planned for and conducted in an in-depth multi-layered unit configuration for both offensive and defensive operations, the use of a remotely operated tethered vehicle with a nearby mother ship acting as the brain is not feasible. Therefore, with the US Navy's current strategy coupled with the Soviet Union's strategy of multi-layered unit configurations, the use of autonomous underwater vehicles (AUV) for potential military operations could significantly improve the US Navy's operational capabilities. These AUVs must possess sufficient onboard sensors as well as control units and be able to perform a preplanned mission without external control. (Robinson 86)

Even with the current decline of the Soviet Union as a world leader due to its internal conflicts and the decline of its satellite countries, the use of AUV technology is still a viable option. This became apparent with the events and operations of DESERT SHIELD and DESERT STORM in the Persian Gulf, especially those dealing with mining of the gulf by the Iraqi naval forces. In addition to using an AUV in mine warfare, an AUV can also be used in many different military operations. Examples of some basic types of operations follow:

1. reconnaissance (both covert and overt) for intelligence gathering
2. surveillance
3. mine warfare
4. terrain mapping
5. supply and resupply for covert units along the coast munitions delivery (or an intelligent torpedo)

The use of an AUV for these type of operations would significantly reduce equipment and personnel to enemy capabilities, thus reducing operational costs no matter the cost of the AUV. Advances in computer technology, particularly in Artificial Intelligence, have made it possible to begin AUV development for these type of missions.

However, concept of traveling from A to B without human interaction requires that the system act and think as a human would. One particular area which offers significant challenges is path planning such that the system avoids all known obstacles within the environment. In a three-dimensional environment, the challenge is even more interesting since there exists an infinite amount of possible paths extending from point A to point B. This concept of three-dimensional path planning is also not limited to an AUV system, but can be used on any system which must traverse a three-dimensional environment.

B. AUV CONCEPT

The basic design of an AUV calls for an unmanned submersible vehicle with onboard systems and sub-systems that provide power, motion control, navigation, obstacle detection and collision avoidance with the capability to perform a preplanned mission by controlling and monitoring onboard systems without any external input. In addition, the AUV must be able to replan its mission in the event it encounters unplanned for events. This includes onboard path replanning to the mission goal in the event the AUV encounters an unplanned-for obstacle (Floyd 91). The Naval Postgraduate School (NPS) AUV II models this basic concept of AUV development; however, an AUV capable of performing the type of missions listed above will need to carry additional equipment tailored to perform the specific mission in order to accomplish the mission.

C. PATH PLANNING AND REPLANNING

The concept behind path planning is to find the shortest path or a relatively short path from an initial starting point to a goal, avoiding all known obstacles in the environment. This concept results in a preplanned mission which is then stored onboard on of the memory modules of the AUV. This path planning process can be accomplished using the AUV control unit, but since this process is a-priori to the mission execution, it would be more feasible to do the

process on another system with more capability, This allows for faster results determining a path, multiple path planning missions and leaves the AUV for other mission executions. However, the replanning concept calls for the path planning process to be performed on the AUV. The concept of an onboard mission planner further implies that the planner must have the capability to access onboard stored information of the environmental model. It further implies that the replanning process must be in strict coordination with not only the obstacle avoidance capability on the AUV, but also the module for location determination. This thesis will address the path planning and replanning process. Work on the obstacle avoidance and associated problems has been addressed by another NPS AUV II project team member (Floyd 91).

D. OBJECTIVES

This thesis will address the following research questions:

1. How to define the typical underwater environment?
2. Given a known underwater world environment, how to plan a route which is the shortest path or a relatively short path and avoids all known polyhedrons in the environment?
3. How to incorporate the path planner as a on-board replanner?

E. THESIS ORGANIZATION

The organization of this thesis is in five chapters. Chapter II contains background information concerning the NPS AUV II system along with a general summary of other AUV projects ongoing as well as research conducted on path planning. Chapters III and IV discuss implementation of the path planning process.

The main idea behind any path planning algorithm is two fold. First, the planner must determine if there exists any obstacles lying between the starting point and the ending or goal point. If there are not any obstacles, then the path from start to goal is the straight line path from start to goal. However, if there is one or more obstacles between the start and goal, then the path planner must begin the planning process to determine the shortest of a relatively short path.

The organization of this thesis follows this idea. Chapter III begins with a discussion of how the three-dimensional model is depicted. Afterwards, the rest of the chapter is devoted to describing the technique used to determine if one or more obstacles lie between the start and goal. Chapter IV describes in detail the path planning process used to determine a best path from start to goal if there is one or more obstacle between them. Finally, Chapter V are my conclusions and recommendations for future research.

II. NPS AUV PROJECT AND RELATED WORK

A. SURVEY OF PREVIOUS WORKS

As stated before, the primary goal of any AUV research is to allow the vehicle to perform a mission without human intervention during the execution of the mission. To operate autonomously, the vehicle must possess adequate intelligence to travel a pre-planned route, but also react to unplanned for situations. There are several ongoing AUV research projects as well as research for path planning for both two-dimensional and three dimensional.

1. Autonomous Underwater Vehicle Projects

Within the past decade, research on Autonomous Vehicle control has made great strides. There are several ongoing projects for AUV development conducted by not only academic institutions, but also government agencies as well.

Texas A&M University is one of the academic institutions working on AUV research. Most project designs not only incorporate the software or an on-board computer controller, but also the specific hardware that the controller controls. Texas A&M University has taken a different approach in the project development. Their approach is to develop a generic computer controller capable of successfully performing a fully autonomous long range

underwater mission. Table I shows the mission requirements. While not specifically building the hardware for the controller to control, a generic vehicle was considered in order to provide design parameters in developing the

Table I - TEXAS A&M MISSION REQUIREMENTS

MISSION REQUIREMENTS
Long Range Capability - 3000 Nautical Miles
Long Duration - 300 Hours
Precise Navigation Capability
Collision Avoidance
Communication Capability with Mother Ship
High Reliability and Adaptability

controller. As can be seen from Table II, the vehicle requirements are unrealistic for any type of testing other

Table II - TEXAS A&M VEHICLE CHARACTERISTICS

VEHICLE CHARACTERISTICS	
Length	81 feet
Diameter	13 feet
Top Speed	12 knots
Cruise	10 knots
Maximum Depth	800 feet
Range	3000 Nautical Miles 300 hours @ 10 NM/Hour
Fuel Load	8.1 tons Diesel
Computational Power	16 SUN Sparc Stations

than simulation testing. Texas A&M researchers use a large vehicle in the development phase with the idea that the generic controller can be adapted for other vehicle designs; however, with the use of such a large computer platform in the design, one major question is whether the controller can be adapted to a platform which does not have 16 SUN Sparc Stations for computational power. (TEXAS A&M 90)

The United States Navy in conjunction with DARPA initiated an AUV program in 1988 with the primary purpose of showing that AUV systems could successfully perform certain Navy mission requirements. Table III shows these mission requirements along with some of the vehicle characteristics. Unlike the Texas A&M project, DARPA is developing a complete system of both hardware and software along with a real-time simulation capability of the AUV. And like the Texas A&M project, DARPA's AUV is a long range platform and not suitable for operations within a small area such as a harbor facility. (Pappas et.al. 91)

Table III - DARPA MISSION AND VEHICLE CHARACTERISTICS

MISSION AND VEHICLE CHARACTERISTICS	
1. Mission	
a. Tactical Acoustic System	
b. Mine Search System	
c. Remote Surveillance System	
2. Vehicle Characteristics	
a. Weight	- 7.5 tons
b. Depth	- 1500 feet
c. Speed	- 10 knots
d. Range	- 240 Nautical Miles 24 hours @ 10 knots

The University of New Hampshire has done extensive research on AUV technology. There initial project began in 1978 with the completion of the first Experimental Autonomous Vehicle (EAVE) and subsequent successful test of autonomously following an underwater pipeline. In 1983 this vehicle successfully followed a pre-defined path using acoustic transponder navigation. In 1986, the university built two new vehicles similar to the original, but with more computational power. Current interest in their research is to use artificial intelligence techniques to develop a knowledge based guidance and control system. (Blidberg 90)

The Institute of Industrial Science at the University of Tokyo has developed their own version of an AUV. The PTEROA150 was built and successfully tested in 1989. After successfully testing this prototype, the university built the PTEROA250 with greater capabilities as well as using neural-net technology for control of the vehicle. The Institutes research has shown that a neural net controller successfully controls the AUV and can be adjusted to allow the vehicle to be used in different environmental conditions. (Ura 90)

2. Path Planning

Articles which present the more important aspects of path planning are presented. To date, there has been a

considerable amount of research on path planning in a two-dimensional model. However, there is limited amounted of research for a three-dimensional model. These articles can be classified into two main categories of path planning. These categories are graph searching techniques and potential field methods. In addition, the majority of articles that do deal with a three-dimensional model find only paths based on the shortest path and do not consider or simplify other constraints such as time allotted for travel, energy consumption, type of mission, and type of environment.

M. Sharir has done extended research in the area of path planning and obstacle avoidance. Sharir presents several algorithms for both two-dimensional and three-dimensional environments (Sharir 87). He and J. T. Schwartz have also collected a group of algorithms on motion planning (Schwartz 88). The algorithms they present are predominantly graph searching techniques and their work catalogs the algorithms into different classes based on each of the algorithms properties. Their work gives a good basis for further research; however, as they point out, they present no research on algorithms based on a best cost property other than shortest euclidean distance since there is very little research ongoing.

In his thesis, Ong describes a heuristic search algorithm that finds a viable path in a three-dimensional

grid coordinate system. Ong uses a graph searching technique and the algorithm, as the name implies, uses detailed heuristic to prune the solution space by selecting only viable successor states for further search. The heuristic that are used allow the vehicle to model the human decision-making process in determining which viable path is followed. Since it uses heuristic extensively, it can be considered as an informed search and gives a semi-optimized solution. It uses a simplified energy consumption rate of 1.2 times the overall length of the path for a path with the start and goal lying at different depths. For a path with turns, the algorithm adds a constant for the size of the turning angle. As the size of the turning angle increases, the larger this cost constant becomes. This is a better estimate than the estimate than the estimate used for depth change; however, it is not accurate. (Ong 89)

Charles Warren uses the potential fields method to determine a viable path. In his research, Warren uses artificial potential fields in a global path planning vice a locally selected region. The idea behind his research is to apply potential fields around configuration space (C-space) obstacles and use these fields to select a safe path. The first step in the process is to map the environment into the C-space of the robot. Then artificial fields are placed around the obstacles in the environment. In a sea environment, many obstacles begin at the ground and extend

up. Since the sea floor is an obstacle as well as these protrusions, the result is one massive obstacle. Warren compensates by creating the potential fields from horizontal planes sliced through the obstacles. Each of these planes forms a forbidden region similar to the two-dimensional problem. Since Warren's method uses a global method, a chosen path is selected and then modified using the potential fields to influence its direction. An advantage to using this technique is that the probability of becoming trapped by a local minimum is greatly reduced since the path planning process uses a known viable path. Warren's research shows a viable and fast path planning process. However, it has its limitations. A major limitation is that the algorithm must know the global environment prior to the process beginning. (Warren 89 and Warren 90)

Martin Herman introduces a fast three-dimensional, collision-free motion planner in his research at the National Bureau of Standards. In his planner, Herman represents the environment in an Octree structure. Using an Octree structure, the environment is initially represented as a single primitive geometric structure, usually a cube. This cube is the root of a octree. Children of a node are created if the parent cubic volume is not homogeneous or completely filled with an object or completely empty. If the cubic volume is not homogeneous, then the cube is subdivided into eight cubes (called octants) with each cube

becoming a child of the parent. Once the environment is represented in an Octree structure, Herman uses one of three search algorithms. They are A*, hill climbing, and hypothesize and test. The path found using one of these algorithms finds a collision-free path, but it is not necessarily the shortest path. As Herman states, finding the shortest path is computational expensive and a relatively short path is adequate for many tasks. One other major limitation to this method of path planning is that considerable memory is required to store the octree. (Herman 89)

Perez and Wesley have introduced in their research, an algorithm that works for both two-dimensional and three-dimensional models. In determining a safe shortest path for an autonomous vehicle, the authors' algorithm grows each obstacle by some parameter that corresponds to the dimensions of the vehicle. With the extra dimensions added to each obstacle, the best path is assured to be a safe path. The algorithm uses a parametric function to allow for vehicles that are not able to change their posture in the environment. The algorithm uses the vertices, represented in the x, y coordinate system, of the obstacle in the two-dimensional environment in finding a shortest path. In the three-dimensional environment, the grown obstacles are represented in the x, y, z coordinate system. However, a shortest path whose node set contains only vertices of these

grown obstacles will not necessarily be the shortest path. As a result, the authors use additional vertices of a constant distance from each other. The resulting path is a good approximation of the optimal path. (Perez et.al. 79)

Kanayama has introduced work on two-dimensional path planning for his ongoing research with the Yamabico Mobile Robot. In his algorithm, Kanayama proposes that fast effective path planning can be accomplished using tangent lines. In using technique, possible paths are determined by determining the tangent lines between the start point and all possible obstacles, between the possible obstacles and other obstacles and finally the obstacles and the goal point (Kanayama 90).

There has been a significant amount of research conducted in pruning or reducing the search space of the visibility graph. Montgomery has introduced several techniques that does such. One of more interest is the technique of finding the farthest points to the left and right with relation to the point being considered for expansion. Once these points are found, Montgomery's algorithm determines the front edge of the obstacle and strips off all vertices that comprise the back edge. He argues that by stripping off the back edges, the computational requirements are greatly reduced since the number of vertices is reduced by roughly a factor of two. The algorithm also strips from the search space any

obstacles that are occluded from the expansion point by another obstacle. (Montgomery et.al. 87)

As can be seen, there is a considerable amount of research on path planning in a three-dimensional environment. Each of the above methods are viable and efficient as a motion planner. However, these methods only use a shortest path (or a relatively shortest path) to determine which path is the best path to use. I found no research which addressed using some other measure to determine a path other than shortest path.

B. THE NPS AUV PROJECT

The NPS AUV PROJECT is currently comprised of two different areas or research. The first area research is computer simulation of the overall project in the laboratories at NPS; the second area is experimental research conducted on the NPS AUV II, using the NPS swimming pool as the world environment.

1. The NPS AUV II Vehicle

The current NPS AUV vehicle is called the NPS AUV II. Its design is based on development of the earlier and smaller prototype, the NPS AUV I (Healey et.al. 89). The basic vehicle design of the NPS AUV II has been detailed by Good (Good 89). Figure 1 illustrates the basic design of the vehicle.

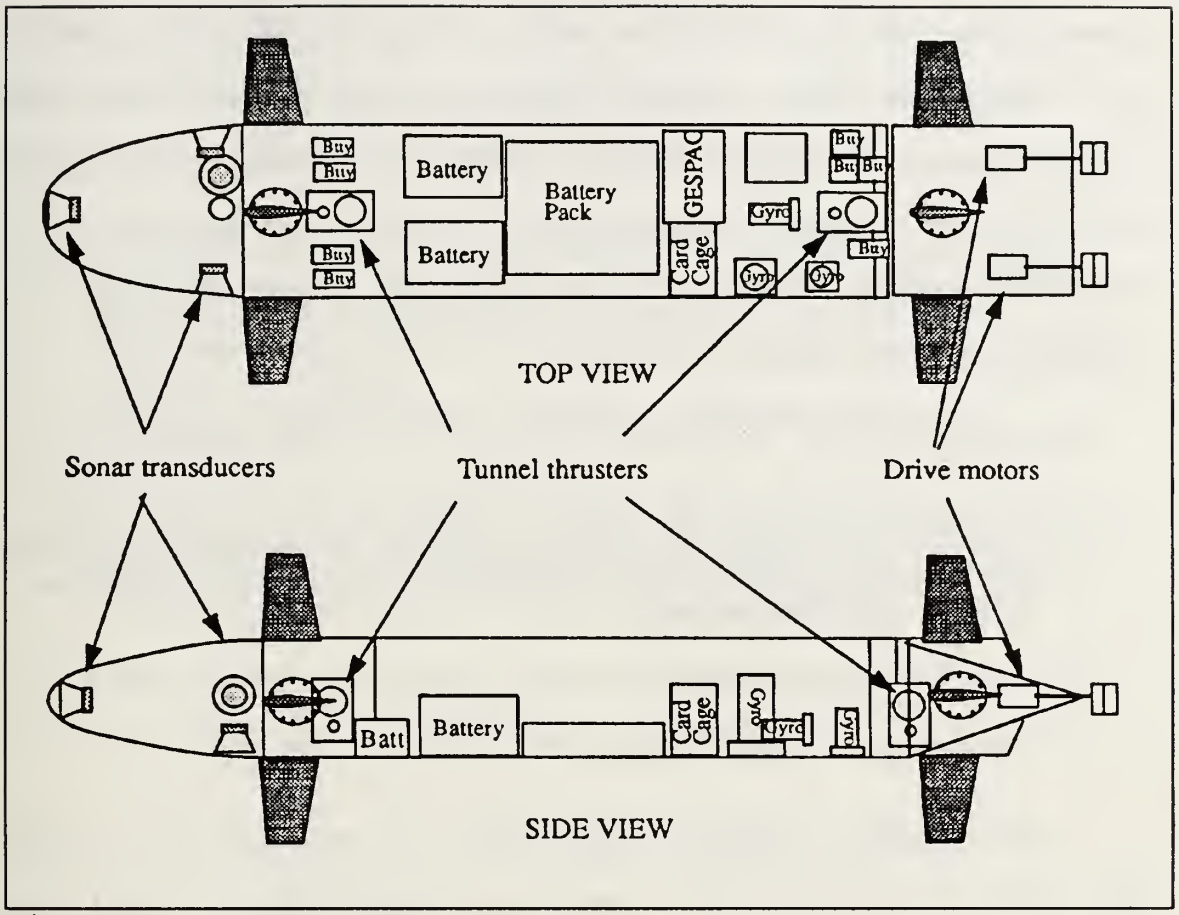

Figure 1 - NPS II AUV

The NPS AUV II is a submersible vehicle capable of a maximum speed of two knots. It has an overall length of 93 inches. The main body of the vehicle is made of aluminum and the constructed as a box. It has a beam of 16 inches, a height of 10 inches and a length of 72 inches. The nose cone is constructed of fiberglass and is 21 inches long, extending the length of the vehicle to 93 inches. The vehicle displaces 380 pounds and uses fixed ballast. The vehicle has four forward control surfaces, four aft control surfaces, four tunnel thrusters and two aft screws each

powered by 24 volt DC drive motors, providing 1/8 horsepower. With the eight control surfaces, the vehicle is highly maneuverable and has a turning radius of 3 ship lengths. The vehicle's power sources are lead-acid gel batteries capable of providing up to two hours of propulsion and computational power.

On board sensors include the following systems:

1. a navigation system sensor comprised of a flux gate compass and directional gyroscope, a vertical gyroscope and a three axis rate gyroscope system with translational accelerometers.
2. a paddle-wheel speed sensor installed in the nose.
3. four sonar transducers installed in the nose.

The on-board computational power is supplied by a GESPAC MPU 30HZ processor with a Motorola 68030 CPU. In addition, the system also has two megabyte of RAM and a 68882 math coprocessor running at 25 MHZ. The operating system is the OS-9 multi-tasking operating system.

2. Computer Simulation

The computer simulation portion of the NPS AUV PROJECT is currently done on a high resolution graphics works-station. The basic simulation model of the NPS AUV II was developed by Jurewicz (Jurewicz 91). This model uses up-to-date performance coefficients characteristic of the NPS AUV II. The basic concept of the simulator is to allow project team members to integrate individual modules of the

project into the simulator for testing prior to implementation on-board the NPS AUV II. The following modules have been implemented on the simulator:

1. A simple locomotion generator for generating a path given a set of reference points along a path. This generator is restricted to a path in which the reference points of the inputted set lying on the same horizontal plane.

2. A three-dimensional guidance control module (Magrino 91)

3. An obstacle detection and avoidance module using sonar technology. This module has also been successfully implemented on the NPS AUV II. (Floyd 91)

III. OBSTACLE REPRESENTATION

One of the first problems I had to solve was determining how to represent each three-dimensional obstacle in the environmental model . There were two areas I had to consider in deciding how to do the representation.

The first area was that I wanted my path planner to use a tangential method similar to the way Kanayama and Crane approach two-dimensional path planning (Kanayama 90 and Crane 91). The other area of concern was that I wanted the path planner to be compatible with not only the NPS pool environment where actual testing of the NPS II AUV is conducted but also the Monterey Bay, which will be included in the simulator at a later date.

In addition to considering these two concerns, I also made some assumptions of the type of obstacles and the type of environment. One such assumption I made concerning the obstacles was that all polyhedrons were considered to be convex in nature. The assumption I made about the environment was that if a path existed between the start and the goal, it was not a path that first retreated in the opposite direction. A path which first retreats in the opposite direction is any path that begins in a direction in the x-y plane which is plus or minus 180 degrees different from the vector from start to goal in the x-y plane.

The bay data was collected in two different resolutions. The first resolution covered a large area and used a distance between depth points of 200 meters; the second resolution, while much smaller gave accuracy with a distance of 30 meters between depth points. With the location and depth known for each point, an XYZ coordinate system could be used in computations. In addition, as with the bay data, an XYZ coordinate system could be used for the NPS Pool and its environment.

With these two concerns in mind, I decided to represent each obstacle in a triply linked list with each obstacle composing the main link, each obstacle-face composing the obstacle link and each vertex of the face composing the obstacle-face link. In addition, the obstacle list comprises first the side faces, followed by the top face if needed, and then the bottom face if needed. Also, each obstacle face list is a doubly linked list and the vertices of the face are placed in a counter-clockwise order. Figure 2 shows the basic concept in this link list structure.

The size of the link list can become quite large especially for a real world environment such as the Monterey Bay. However, the size of this link list can be pruned once the user inputs the start point and the goal point. For example, once the user inputs these points, the algorithm can prune from the list all obstacles which lie behind the start goal.

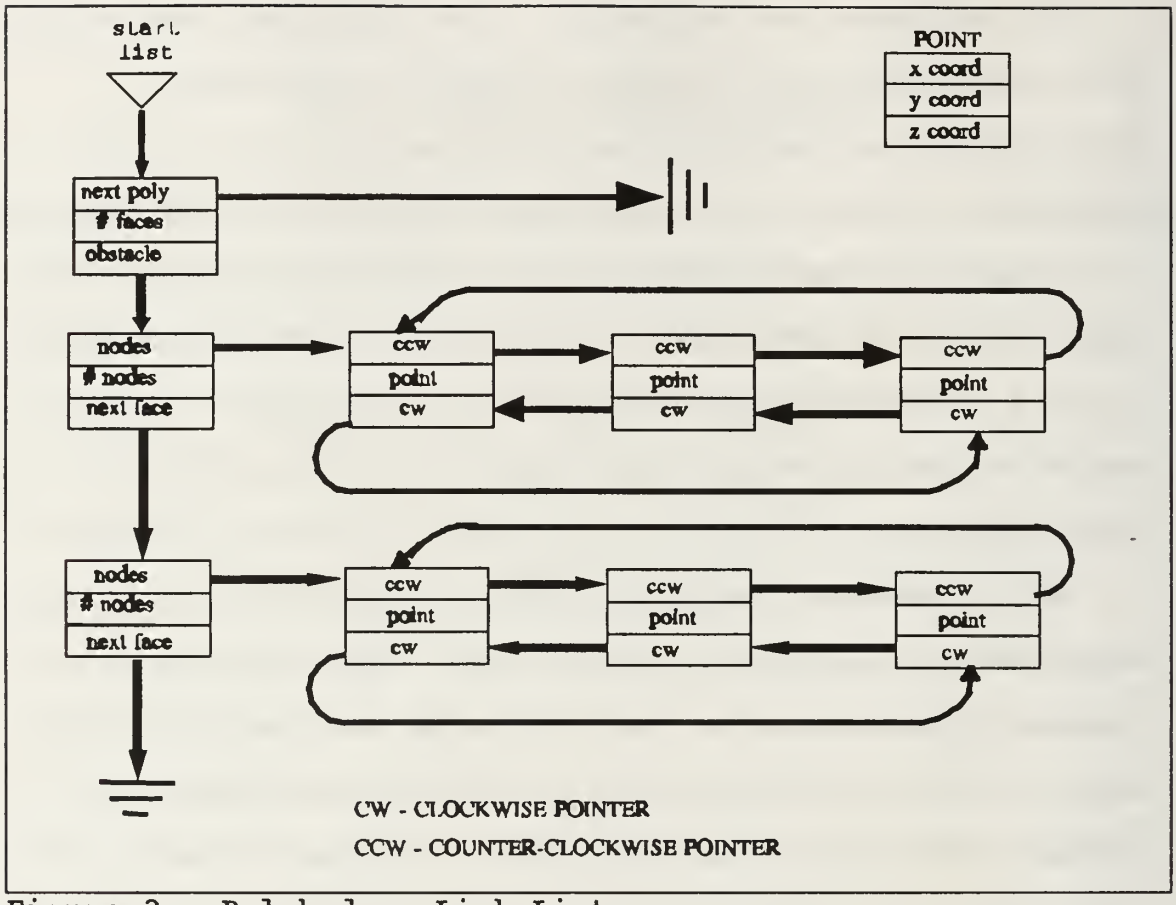

Figure 2 - Polyhedron Link List

IV. PATH PLANNING

If the visibility check between the start and goal points determines that there are obstacles lying between the two points, the path planning process begins. With path planning in a two-dimensional environment, the process is relatively simple since all paths are on the same plane or surface. However, three-dimensional path planning does not have the same characteristics. The three-dimensional environment has an infinite amount of planes and possible paths to consider. As a result, I had to develop a process which reduced the search space and the number of possible paths to consider to a reasonable amount. In developing the algorithm, I considered only obstacles which rise from the floor of the environment and are convex. By considering only these type of obstacles, I do not have to consider a possible path which goes underneath an obstacle. Even though, as Sharir states, the problem still approaches polynomial time in order of magnitude (Sharir 87).

A. BASIC CONCEPTS OF THREE DIMENSIONAL PATH PLANNING

In developing an algorithm to find a best path in a three dimensional environment, I decided to extend Kanayama's two-dimensional path planning algorithm. Kanayama's technique develops tangents from the expansion

point to each valid two-dimensional polygon in determining a best path (Kanayama 90). In addition to using expanding on his technique, I will also expand on Kanayama's terminology.

1. Polyhedron Intersection

In order to solve the problem of determining whether or not a polyhedron lies between an expansion point and the goal, I used vector calculus and developed a simply ray tracing algorithm to determine visibility. A point is visible with another point if and only if no polyhedrons intersect the vector formed by the two points. This process of determining if two points are visible or not is not only used at the beginning of the path planning process with the start and goal points used, but also each time the path planning process expands a path. Therefore, for explanatory reasons, I use the term "Expanding Point" and its variations in the explanation to denote the first point used. Similarly, I use "End Point" to denote the point the process is expanding to.

a. General Description.

The process initially finds the ray formed by the expansion point and the goal. From the ray, the process determines the vector. Once this vector is found, the process finds the equation of the plane formed by the first face of the first polyhedron in the list of polyhedrons, determines if there exists an intersection between the

expansion-goal ray and the plane, and if one exists, determines if the intersection point lies within the boundaries of the polyhedron face. If the intersection point lies between the boundaries, the process ends notifying the calling process that a polyhedron lies between the two points. Otherwise, the process continues checking each face of each polyhedron, notifying the calling process that the two points are visible to each other. Table IV depicts the pseudo language I developed for the process.

Table IV - PSEUDO LANGUAGE FOR INTERSECT POLYHEDRON

```
intersect_polyhedron(pt1, pt2, 3D_world)
{
  for(each polyhedron in 3D_world){
    for(each face on polyhedron){
     plane = find_plane_equation(3 points on face);
     intersection_pt = find_intersection_point(pt1, pt2,
                                                plane);
      if(intersection_pt != NO_INTERSECTION){
        if(lines_intersection(intersection, face, plane)
          return(INTERSECT);
      }
    }
  }
  return(NO_INTERSECT):
}
```

b. Directed Ray.

The process begins by determining the parametric equations for the ray formed using the expansion point and the goal. I directed the ray from the expansion point and the goal so the equations take on the form in Equation 1.

$$x = x_e + (x_g - x_e)t$$
$$y = y_e + (y_g - y_e)t \qquad \text{Eq 1}$$
$$z = z_e + (z_g - z_e)t$$

However, in this case, the direction of the ray does not matter and adopted this manner for consistency and readability.

c. Plane Equation.

In determining the equation of a plane formed by a polyhedron face, the process uses 3 points $\{(x_1,y_1,z_1), (x_2,y_2,z_2), (x_3,y_3,z_3)\}$ from the polyhedron face the process is checking. The process uses these 3 points to determine 2 vectors in the form of Equations 2 and 3.

$$\boldsymbol{V_1} = (x_2 - x_1)\boldsymbol{I} + (y_2 - y_1)\boldsymbol{J} + (z_2 - z_1)\boldsymbol{K} \qquad \text{Eq 2}$$

$$\boldsymbol{V_1} = (x_3 - x_1)\boldsymbol{I} + (y_3 - y_1)\boldsymbol{J} + (z_3 - z_1)\boldsymbol{K} \qquad \text{Eq 3}$$

Once these two vectors are found, the process finds the normal vector to the plane by determining the dot product of the two vectors as seen in Equation 4.

$$\boldsymbol{N} = \boldsymbol{V_1} \cdot \boldsymbol{V_2} \qquad \text{Eq 4}$$

N also takes on the form

$$\boldsymbol{N} = n_1\boldsymbol{I} + n_2\boldsymbol{J} + n_3\boldsymbol{K} \qquad \text{Eq 5}$$

Using Equation 5, the equation of the plane can be found by substituting a point on the plane into the equation. If the point (x_1,y_1,z_1) is used, the result is Equation 6.

$$n_1(x - x_1) + n_2(y - y_1) + n_3(z - z_1) = 0 \qquad \text{Eq 6}$$

Equation 6 equates to Equation 7

$$n1x + n_2y + n_3z - D = 0 \qquad \text{Eq 7}$$

where D equates to Equation 8.

$$D = n_1x_1 + n_2y_2 + n_3z_3 \qquad \text{Eq 8}$$

The source code I developed to find the plane equation is shown in Table V and in Appendix C.

Table V - FIND PLANE EQUATION SOURCE CODE

```
equation         find_plane_equation(pt1, pt2, pt3)
   point            pt1, pt2, pt3;
{
   point            vector1_2, vector1_3;
   equation         pl_eq;

   vector1_2.x = pt2.x - pt1.x;
   vector1_2.y = pt2.y - pt1.y;
   vector1_2.z = pt2.z - pt1.z;
   vector1_3.x = pt3.x - pt1.x;
   vector1_3.y = pt3.y - pt1.y;
   vector1_3.z = pt3.z - pt1.z;

   /*cross product is the standard cross product
     equation.  Function also is in 3dd.c*/
   pl_eq = cross_product(vector1_2, vector1_3);
   pl_eq.d = ((pl_eq.x * (-1.0) * pt1.x) +
             (pl_eq.y * (-1.0) * pt1.y));
   pl_eq.d = pl_eq.d + (pl_eq.z * (-1.0) * pt1.z);
   return (pl_eq);
}
```

d. Intersection Point.

Since the process needs to find the intersection point formed by the ray and the plane, the values for x, y, and z used in Equation 1 can be substituted for x, y, and z in Equation 7. This results in Equation 9.

$$n_1(x_e + (x_g - x_e)t - x_1) + n_2(y_e + (y_g - y_e)t - y_1) + n_3(z_e + (z_g - z_e)t - z_1) = 0 \qquad \text{Eq 9}$$

Finally, Equation 10 results when t is solved.

$$t = -\frac{n_1x_e + n_2y_e + n_3y_e + D}{n_1(x_g + x_e) + n_2(y_g + y_e) + n_3(z_g + z_E)} \qquad \text{Eq 10}$$

If we let w and Y take the form of Equations 11 and 12, then t equates to Equation 13

$$w = n_1x_e + n_2x_e + n_3y_e + D \qquad \text{Eq 11}$$

$$y = n_1(x_g - x_e) + n_2(y_g - y_e) + n_3(z_g - z_g) \qquad \text{Eq 12}$$

$$t = -\frac{w}{y} \qquad \text{Eq 13}$$

In equating t to Equation 13, there is a check in the value of y. By first computing y and determining if y = 0, then the ray is parallel to the plane, no other computations are necessary, and the function returns to the calling function that there is no intersection for the checked plane face. If y != 0, then the process finds the value for the parameter w.

If $0 \leq t \leq 1$, then the ray intersects the plane between the two points. If $t < 0$ or $t > 1$, then the ray intersects the plane, but not on the line segment formed by the two points. If the process determines there is an intersection, the value of t is used in Equation 1 to calculate the intersection point and the process returns the point to the calling function. Table VI shows the source code I developed for the process. The source code is also

in Appendix C, which contains the source code of functions which deal with the three-dimensional environment,

Table VI - INTERSECTION POINT SOURCE CODE

```
point   intersection_point(point pt1, point pt2,
                           equation plane)
{
   double          y = 0.0, t = 0.0, w = 0.0;

   y = (plane.x * (pt2.x - pt1.x) + plane.y *
       (pt2.y - pt1.y) + (plane.z * (pt2.z - pt1.z));

   if (!y) {    /* y == 0*/
      intersection.x = NO_INTERSECTION;
   }
   else{
     w = -(plane.x * pt1.x + plane.y * pt1.y + plane.z *
           pt1.z + plane.d);
     t = w / y;

     if (0.0 <= t && t <= 1.0) {
        intersection.x = pt1.x + (pt2.x - pt1.x) * t;
        intersection.y = pt1.y + (pt2.y - pt1.y) * t;
        intersection.z = pt1.z + (pt2.z - pt1.z) * t;
     }
     else
        intersection.x = NO_INTERSECTION;
   }
   return (intersection);
}
```

e. Inside Polyhedron Check.

If the process determines that there is a valid intersection point, it then determines if the point lies within the boundaries of the polyhedron. To determine this, the process projects a ray from the point to a point outside the search space, but lying on the same x y z plane as the

polyhedron face. The process then determines the parametric equations of this ray. Next, by finding the parametric equations for each boundary segment of the polyhedron face, we can equate the two sets of parametric equations. Since there are two unknowns (both parameters), the process uses two of the three equations to simultaneously solve for the unknowns.

For example, if the points of the ray are (u_1, v_1, w_1) and (u_2, v_2, w_2), the parametric equations are shown in Equation 14.

$$\begin{aligned} x &= u_1 + (u_2 - u_1)s \\ y &= v_1 + (v_2 - v_1)s \\ z &= w_1 + (w_2 - w_1)s \end{aligned} \qquad \text{Eq 14}$$

Similarly, if the two end nodes of the boundary of the polyhedron face are (a_1, b_1, c_1) and (a_2, b_2, c_2), the results are shown in Equation 15.

$$\begin{aligned} x &= a_1 + (a_2 - a_1)s \\ y &= b_1 + (b_2 - b_1)s \\ z &= c_1 + (c_2 - c_1)s \end{aligned} \qquad \text{Eq 15}$$

If we use the first two equations of Equations 14 and 15, we can solve for s and t. Equations 16 and 17 depict the results.

$$t = \frac{((b_1 - v_1)\ (a_2 - a_1)) - ((a_1 - u_1)\ (b_2 - b_1))}{((a_2 - a_1)\ (v_2 - v_1)) - ((b_2 - b_1)\ (u_2 - u_1))} \qquad \text{Eq 16}$$

$$s = \frac{((b_1 - v_1)\ (u_2 - u_1)) - ((a_1 - u_1)\ (v_2 - v_1))}{((a_2 - a_1)\ (v_2 - v_1)) - ((b_2 - b_1)\ (u_2 - u_1))} \qquad \text{Eq 17}$$

If $0 \leq s \leq 1$ and $0 \leq t \leq 1$ are valid statements, then the ray intersects the polyhedron face. This process

must be done for each boundary of the face with a counter counting the number of times the ray intersects the boundary, If the counter is an even number, then the intersection point is outside of the boundaries of the face and thus the projection from the expansion point to the goal does not intersect the polyhedron face. Conversely, if the count is odd, the start goal ray intersects the polyhedron face.

An easy implementation of determining if the counter is odd or even is to use modulo arithmetic. The pseudo language in Table VII depicts this process. The source code can again be found in Appendix C.

Table VII - PSEUDO LANGUAGE FOR POLYHEDRON INTERSECTION

```
if (line_intersection_count modula 2 == 1) then
          polyhedron_intersection = True
 else
          polyhedron_intersection = False
end if
```

As mention above, the process checks each face of each polyhedron. If there is not an intersection between the vector and any of the polyhedron faces, the calling process is notified by passing it a non-intersection flag. However, if there is an intersection, the process stops checking the remaining faces and notifies the calling

process of the intersection so that the path planning process can begin.

2. Plane Types

The use of the plane is used several times throughout the process of finding the best path. There are two different types of planes used in finding the best path. I refer to one of the two types of planes used as a "vertical plane". The other plane is the "horizontal plane". I refer to a vertical plane as such because two of the three points lie on the same X Z plane. I refer to a horizontal plane as such because two of the three points are on the same X Y plane and it is perpendicular to the vertical plane formed by the same expansion and end points. The process of finding the equation for both types are the same and uses the same process described earlier in this chapter. Appendix C contains the source code which finds both of these planes.

3. Two-Dimensional Polygons

Two-dimensional polygons are the type of obstacles which are used primarily for the path planning process. The general concept in building the two-dimensional world, the Build_two_d_polygon_list function, cycles through each face of each polyhedron, finding the intersection points of the horizontal plane and the line segments of the polyhedron face if one exits. If an intersection does exist, then the

process puts the point into a link list for the two-dimensional list. Figure 3 depicts the structure of this link list and Table VIII shows the pseudo language used to develop the list. The source code can be found in Appendix B.

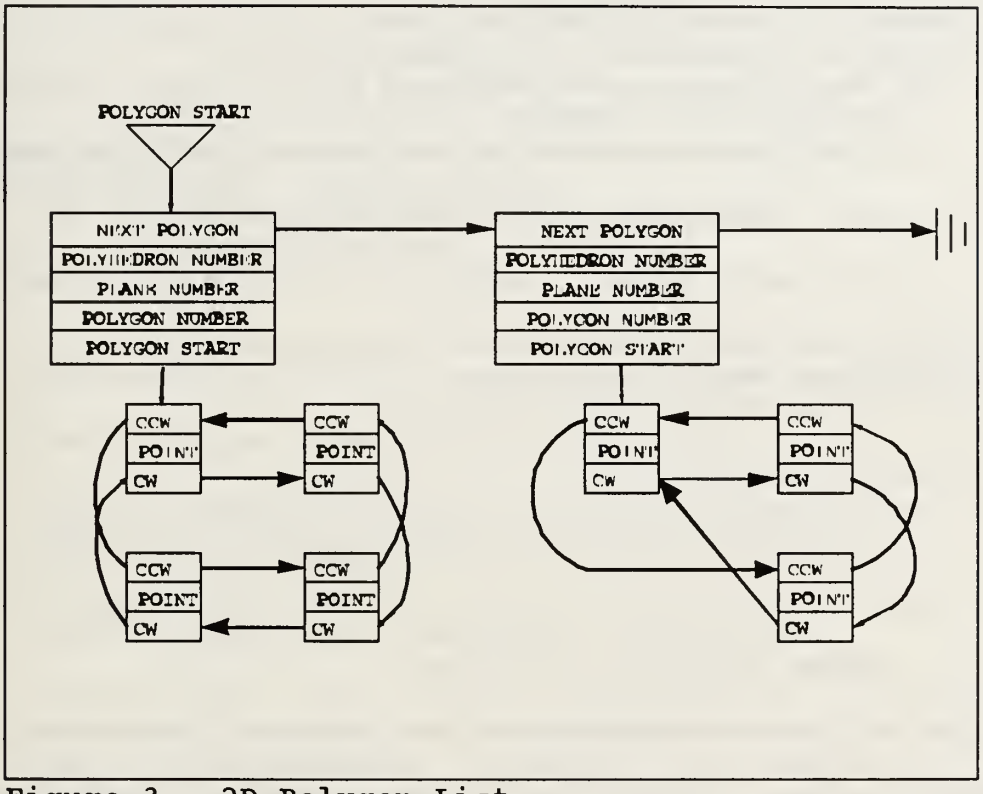

Figure 3 - 2D Polygon List

The Next Polygon entry is a pointer which points at the next polygon in the linked list. The Polyhedron Number entry is the number of the polyhedron which the algorithm used to find the polygon. The Polygon Number entry is the number associated to when the polygon was formed. The Polygon Start entry is a pointer which points to the first node or vertex of the polygon. The nodes are in a doubly

linked list with "ccw" depicting counter clockwise and "cw" meaning clockwise in order.

Table VIII - BUILD TWO-DIMENSIONAL POLYGON

```
Build_two_d_polygon_list(PolyhedronList, Plane)
{
  initialize first_polygon;
  current_polygon = first_polygon
  for(Each Polyhedron in PolyhedronList){
    for(Each Face on Polyhedron){
      for(Each Line Segment of the Face){
         if((intersection = intersection_point(node1,
                        node2, plane) != NO_INTERSECTION)
             Create_polygon(intersection,
                            current_polygon);
      }
    }
  remove_duplicate_node(current_polygon);
  initialize next_polygon;
  current_polygon = next_polygon;
  }
  return(polygon_list);
}
```

4. Number and Types of Paths

In the two-dimensional environment, the number of possible paths is 2^n where n is the number of obstacles in the environment. As already stated, I do not consider paths which go underneath obstacles. As a result, the number of possible paths in the three-dimensional environment is 3^n. In using Kanayama's terminology where a plus sign (+) denotes a path extending counter clockwise around an obstacle and a minus sign (-) denotes a path extending clockwise, I extend it to include an up arrow (↑) which

denotes a path extending over the obstacle. Table VII shows the possible paths for an environment with 1, 2 and 3 obstacles.

In analyzing these possible paths in Table IX, one can see that general paths with no up arrows indicate that the path lies on the same plane as the start and goal. In addition, there is one path which extends above all the obstacles. Thus, all other paths are a combination of the two. In further analysis of Table IX, these paths can be generalize to include all cases. These cases are:

1. The path lies on the same plane as the start and goal.
2. The path goes over all the obstacles.
3. The path goes over one or more obstacles before going around the remaining one or more obstacles.
4. The path goes around one or more obstacles before going over the remaining one or more obstacles.
5. The path goes over one or more obstacles, goes around one or more obstacles and then goes over the remaining one or more obstacles or begins the sequence of around and back over the remaining obstacles until the path reaches the goal.
6. The path goes around one or mo e obstacles, goes over one or more obstacles and then goes around the remaining one or more obstacles or begins the sequence of over and back around the remaining obstacles until the path reaches the goal.

I could argue that bullets 3 and 4 are subsets of bullets 5 and 6 respectively. However, the concept of three-dimensional path planning is a difficult one. Therefore, I have broken the types of paths as above,

allowing me to explain in a more forthright and detailed manner the process I developed to find the best path.

Table IX - POSSIBLE PATHS FOR POLYHEDRON WORLD

1 OBSTACLE (A)	2 OBSTACLES (A, B)	3 OBSTACLES (A, B, C)
A+	A+B+	A+B+C+
A-	A+B-	A+B+C-
A↑	A+B↑	A+B+C↑
	A-B+	A+B-C+
	A-B-	A+B-C-
	A-B↑	A+B-C↑
	A↑B+	A+B↑C+
	A↑B-	A+B↑C-
	A↑B↑	A+B↑C↑
		A-B+C+
		A-B+C-
		A-B+C↑
		A-B-C+
		A-B-C-
		A-B-C↑
		A-B↑C+
		A-B↑C-
		A-B↑C↑
		A↑B+C+
		A↑B+C-
		A↑B+C↑
		A↑B-C+
		A↑B-C-
		A↑B-C↑
		A↑B↑C+
		A↑B↑C-
		A↑B↑C↑
TOTAL POSSIBLE PATHS		
3	9	27

5. Tangents

As with Kanayama's two-dimensional path planning algorithm, my process uses tangent lines extended from the start point or an expansion point to all valid obstacles. However, my algorithm extends these tangent lines to not only the sides of the obstacle, but also to the top of the obstacle.

These two basic tangent types the process uses in the path planning process can be classified as either horizontal or vertical tangents. I further sub-classify the horizontal tangent as either a plus tangent or a minus tangent.

a. Plus and Minus Tangents

I define a tangent to a three-dimensional polyhedron to be the ray which is formed by the expansion point and an intersection point on an edge of the polyhedron such that the ray intersects the polyhedron only at the intersection point. For the case where a tangent crosses a face of a polyhedron, then the first intersection point is the considered the tangent point. As in Kanayama's algorithm for the two-dimensional environment, I define a plus tangent to be a tangent which if it was extended around the obstacle so that it encircled the obstacle, it would wrap around the obstacle in a counter-clockwise direction. Conversely, a minus tangent is a tangent which if the

tangent encircled the obstacle, it would wrap around the obstacle in a clockwise direction. Figure 4 shows the different plus tangents for a two-dimensional polygon. Similarly, Figure 5 shows the different minus tangents.

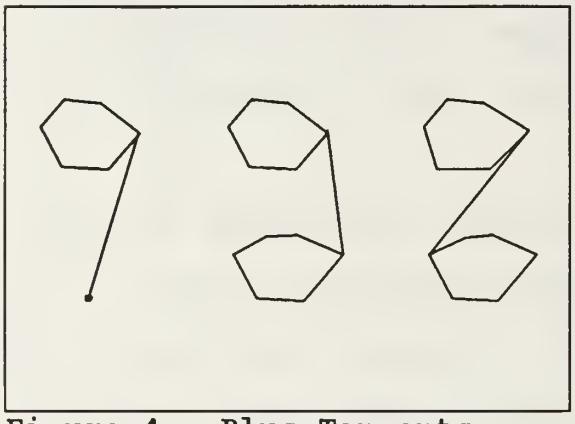

Figure 4 - Plus Tangents

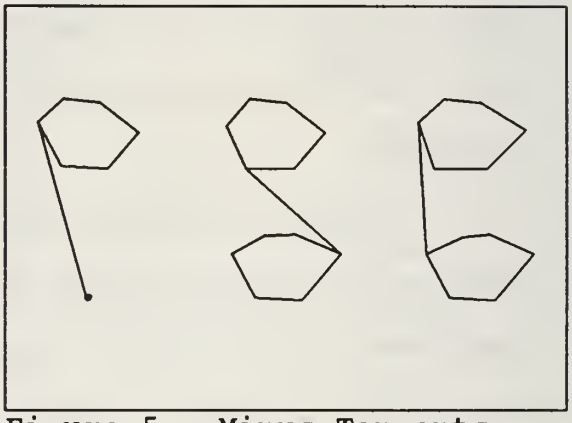

Figure 5 - Minus Tangents

Up until now, I have shown the plus and minus tangents for a two-dimensional polygon. But as Figures 6 and 7 show, there are an infinite amount of plus and minus

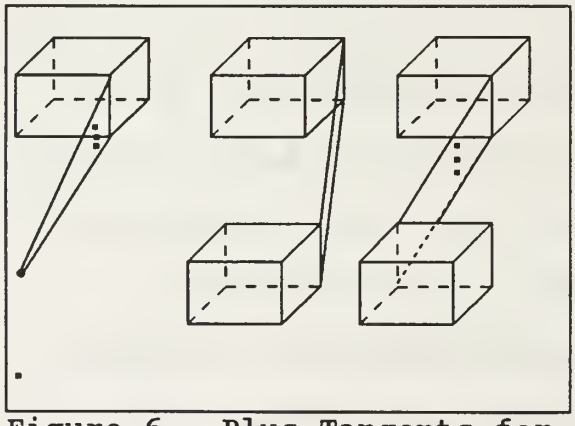

Figure 6 - Plus Tangents for Polyhedron World

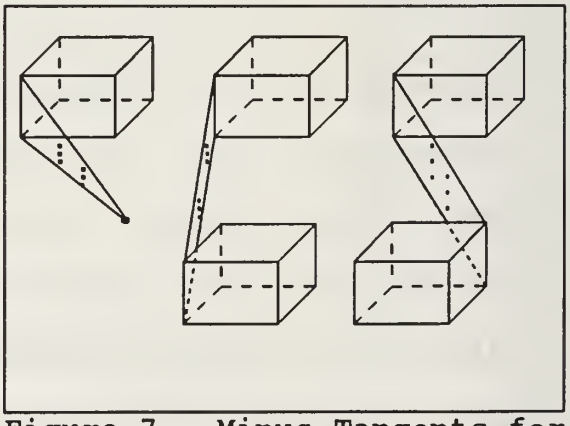

Figure 7 - Minus Tangents for Polyhedron World

tangents when dealing with a three-dimensional environment. However, this unmanageable amount can be reduced to a more manageable amount determined by the type of path being

expanded on and the number of polyhedrons within the environment.

The general idea behind this reduction is to find the polygon formed by the intersection of a plane and the polyhedron. By using this plane to find all the polygons formed by the intersection of the plane and the polyhedrons in the environment, the three-dimensional world is reduced to a two-dimensional representation. Of course, the number of plane slices needed will be different for each of the types of paths as listed above. See Figure 8 for a graphical representation of this intersection.

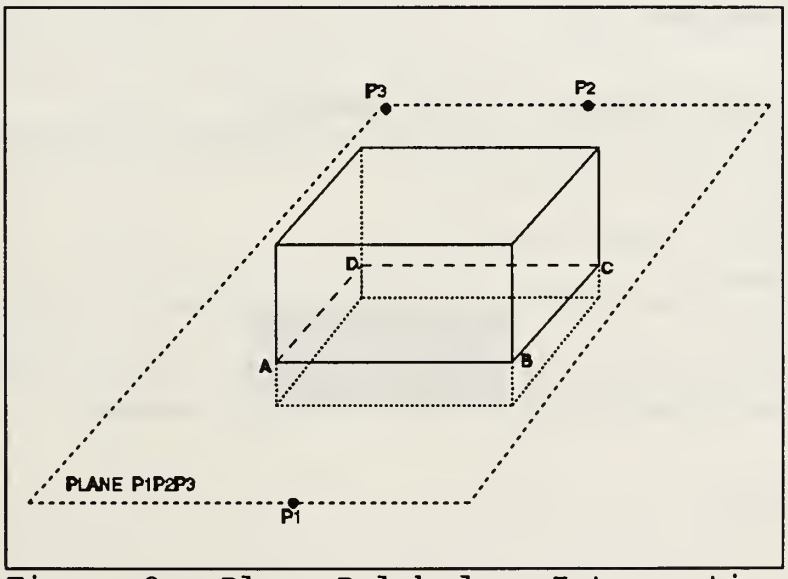

Figure 8 - Plane Polyhedron Intersection

I use Kanayama's basic algorithm for finding each of the type of tangents from a point to a polygon. However, I expand on it to find only the closest tangent point to the expansion point. This case must be accounted for when the

expansion point and one of the sides of the polygon lie on the same projected ray. Table X shows the pseudo language Kanayama has developed to find the plus tangent (Kanayama 90). The minus tangent code is the same except that the "if" structure which checks the sign and order is evaluated with -1 and not 1. The sign function accepts an integer which the order function returns and returns either a 1 or -1 based on whether or not the function is positive or negative. The order function is a modification of the area of a triangle equation. The function determines the order, either clockwise or counter-clockwise, of the three points. For further discussion, see Kanayama's development of the function (Kanayama 90).

Table X - PLUS TANGENT PSEUDO LANGUAGE

```
plus_tangent(pt, polygon_A)
{
  q = initial_node(polygon_A);
  doforever{
    if(sign(order(pt, q, next(q)) == 1) q = next(q);
    else{
      if(sign(order(pt, q, prev(q)) == 1) q = prev(q);
      else break;
    }
  }
  return(q);
}
```

In determining if the expansion point lies on the same project ray as a side of a polygon, the process must make two checks. The first check determines if the

direction of the ray formed by the expansion point and the point found by the plus or minus tangent function is equal to the ray formed by the expansion point and the vertex node counter-clockwise to the expansion point. Similarly, the second check determines if the ray formed by the expansion point and the vertex node clockwise to the point is equal to the ray formed by the tangent point and expansion node. If the rays are equal, then the vertex node which is closest to the expansion point is determined and returned. Figure 9 shows these scenarios. Table XI shows excerpts of the C code I developed to accomplish this. Appendix I gives the complete source code for both the plus and minus tangent functions.

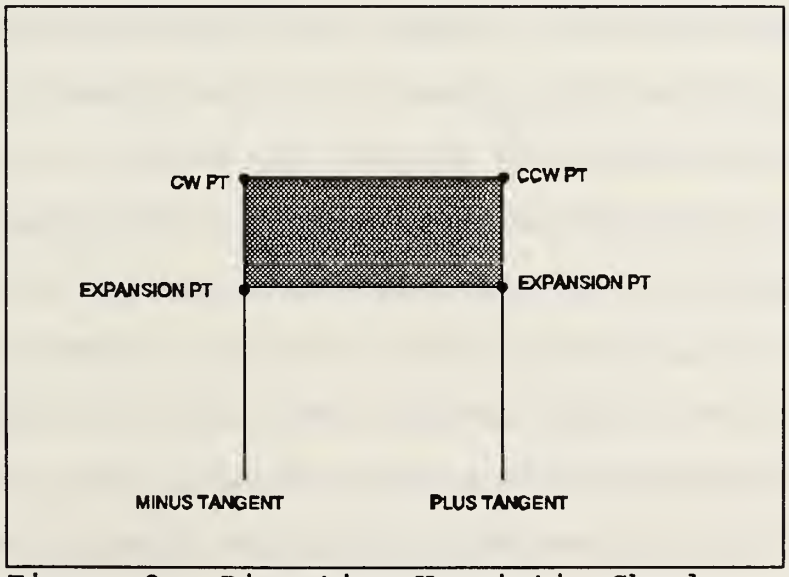

Figure 9 - Direction Heuristic Check

Table XI - DETERMINING CLOSEST TANGENT POINT SOURCE CODE

```
if(atan2(p1.y - node->pt.y, p1.x - node->pt.x) ==
   atan2(p1.y - node->ccw->pt.y, p1.x - node->ccw->pt.x)){
  if(distance(p1, node->pt) <
        distance(p1, node->ccw->pt))
    return (node->pt);
  else
    return (node->ccw->pt);
}
if(atan2(p1.y - node->pt.y, p1.x - node->pt.x) ==
   atan2(p1.y - node->cw->pt.y, p1.x - node->cw->pt.x)){
  if(distance(p1, node->pt) <
        distance(p1, node->cw->pt))
    return (node->pt);
  else
    return (node->cw->pt);
}
return (node->pt);
```

The case of finding the tangent from one polygon to another polygon needs to be handled in a different manner. The problem encountered in finding this type of tangent happens when the tangent node from the first polygon is not the point on the polygon which the process is expanding from. Figure 10 depicts this case.

As can be seen in Figure 10, if the plus tangent is found using the expansion point, the tangent line (Expansion Point, A) cuts across the first polygon, which cannot occur for a valid partial path. The same thing occurs when trying to find a minus tangent. As a result, I had to develop a process which found the appropriate tangents between the polygons (tangent lines (C,B) and (E,D) respectively).

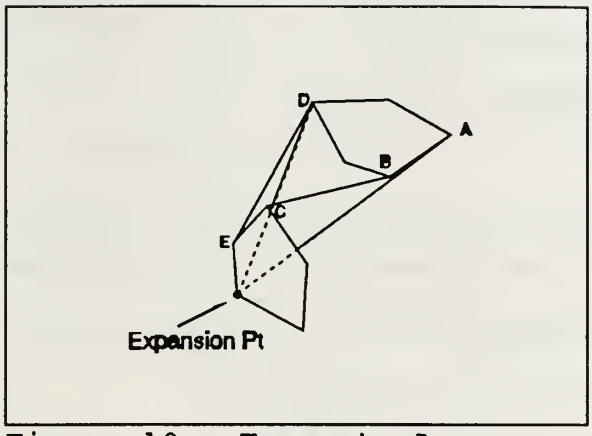

Figure 10 - Tangents Among Polygons

In finding the two appropriate tangent nodes on the polygons, I had to consider what type of tangent was used in arriving at the expansion point. The idea behind the process is then to find the tangent node on the second polygon from the expansion point. This tangent node is then used in finding the tangent node on the first polygon. This process of oscillating from one polygon to the next polygon continues as long as the tangent line intersects one of the two polygons. Upon finding the two points the process returns the two nodes in a link list to the calling function. Table XII depicts this process in pseudo language for each of the different cases in finding the appropriate tangent. Appendix I contains the function "tangent" written in the C language, which also has all the cases.

Table XII - FINDING APPROPRIATE TANGENTS

```
polygon1_pt = expansion_pt;

/*expansion pt tangent is minus
  plus tangent to polygon2*/
do {

  polygon2_pt = plus_tangent(polygon2, polygon1_pt);
  polygon1_pt = plus_tangent(polygon1, polygon2_pt);

}while (not(visible(polygon1_pt, polygon2_pt);

/*expansion pt tangent is minus
  minus tangent to polygon 2*/
do {

  polygon2_pt = minus_tangent(polygon2, polygon1_pt);
  polygon1_pt = plus_tangent(polygon1, polygon2_pt);

}while (not(visible(polygon1_pt, polygon2_pt);

/*expansion pt tangent is plus
  plus tangent to polygon 2*/
do {

  polygon2_pt = plus_tangent(polygon2, polygon1_pt);
  polygon1_pt = minus_tangent(polygon1, polygon2_pt);

}while (not(visible(polygon1_pt, polygon2_pt);

/*expansion pt tangent is minus
  minus tangent to polygon 2*/
do {

  polygon2_pt = minus_tangent(polygon2, polygon1_pt);
  polygon1_pt = minus_tangent(polygon1, polygon2_pt);

}while (not(visible(polygon1_pt, polygon2_pt);
```

b. Vertical Tangents

Compared to finding the plus and minus tangents, finding the vertical tangents is a relatively simple exercise. Since I only am concerned about the top vertical

tangent of each polyhedron, I refer to this tangent as the vertical tangent. Figure 11 depicts the vertical plane, the obstacle, and the tangent.

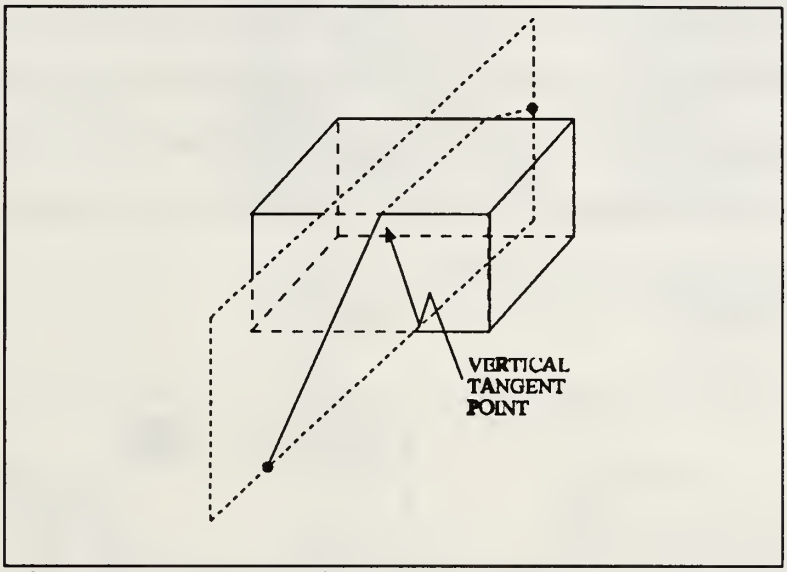

Figure 11 - Vertical Tangent

Unlike the functions plus_tangent and minus_tangent, which find a point which forms the tangent, the process I developed finds not only that one point which forms the tangent, but it also finds the intersection point of the vertical plane which is opposite to the side of the top polyhedron face from which came the first point. Figure 12 depicts the scenario. I include both intersection points because if the vertical extension of a path is to go over only one polyhedron, then the second point will eventually be needed. Since the vertical plane is already determined, then it is logical to expand the path to the second point and not the first. In addition, the process finds all vertical points to all polyhedrons lying between the

expansion point and the end point. In the event that there are more than one obstacle, then I have also included a check which determines if points are visible not to the next point in the list, but to the other points in the vertical list as depicted in Figure 13. This allows me to exclude all intersection points not needed in developing the extended path. Appendix J contains the source code for finding these vertical points.

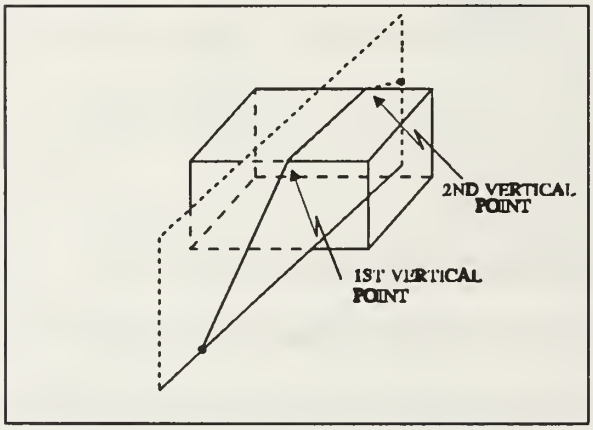

Figure 12 - Vertical Points for One Polygon

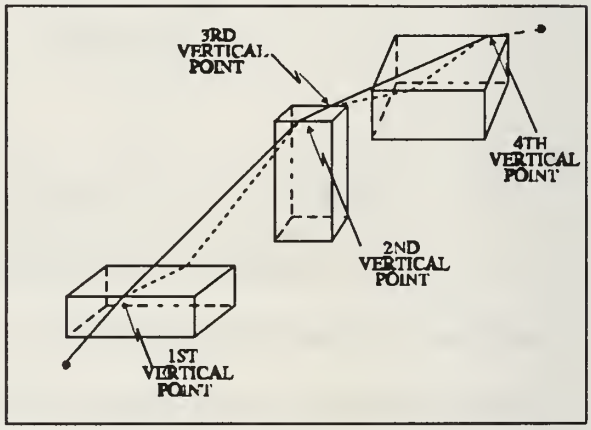

Figure 13 - Vertical Nodes for More Than One Polygon

6. Path Representation

In designing my algorithm for the path planning process, I had to develop a structure which I could represent the path. This path structure evolved through many iterations until I arrived at the final structure. Figure 14 shows the structure and the file 3d_tan.h is in Appendix L contains the source code defining the structure.

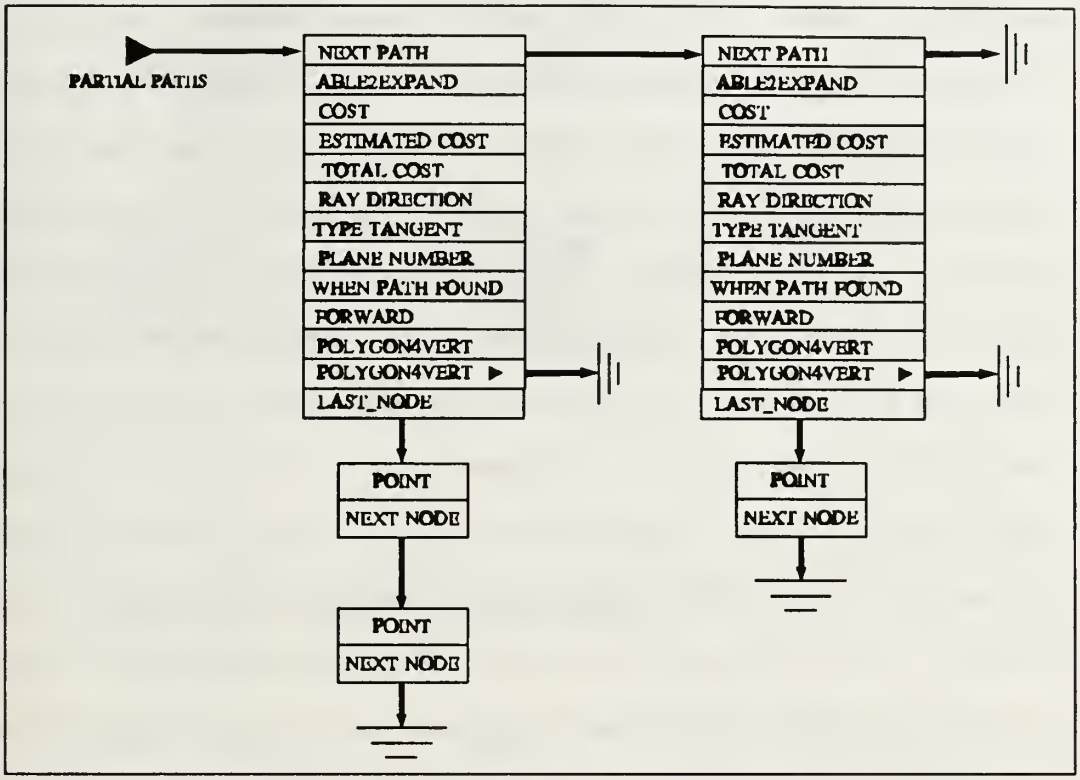

Figure 14 - Partial Path List

The Next_path pointer and the Last_node pointer in the list are self-explanatory. The 3 cost entries are used in the search function to identify the shortest path to expand the path. The cost entry is the cost from the start point traveling along the path to the last node added to the goal; the estimated cost entry is the euclidean distance from the last point added to the goal; and the total cost is the summation of the other two costs. The ray direction entry is the direction on the X Y plane of the tangent from the second to last node to the last node added to path. This entry is used in a direction heuristic, which is used to prune possible path entries. The type_tangent entry is the type of the last tangent formed by the last node and the

second-to-last node. The entry "Plane number" is the number of the horizontal plane slice used in finding the path around obstacles. This entry ensures that the same plane is used to expand a path around the obstacles. The able2expand entry is a flag used by one of the heuristic to determine if the path should be extended or not. The integer forward identifies the path as built from the start to goal or from the goal to the start. Polyhed4vert identifies the polygon which the path will go around after going over the preceding obstacles. The Polyhed4vert_PTR is a pointer which points to an X Y node structure and is used to expand the path from a vertical node around an obstacle and over other obstacles (path types 5 and 6). I refer to the information in this structure as the header information of the path.

7. Active Node List and Active Paths Heuristic

The Active Node List is a link list which identifies all those nodes which have been added to the paths in the partial path list. Figure 15 shows the structure I use in this link list. As with the partial path structure, the source code for this structure is found in 3d_tan.h in Appendix L. Along with the X Y Z point, the structure also has an entry for the cost of the path from start to this X Y Z point and an integer which corresponds to when the path was found.

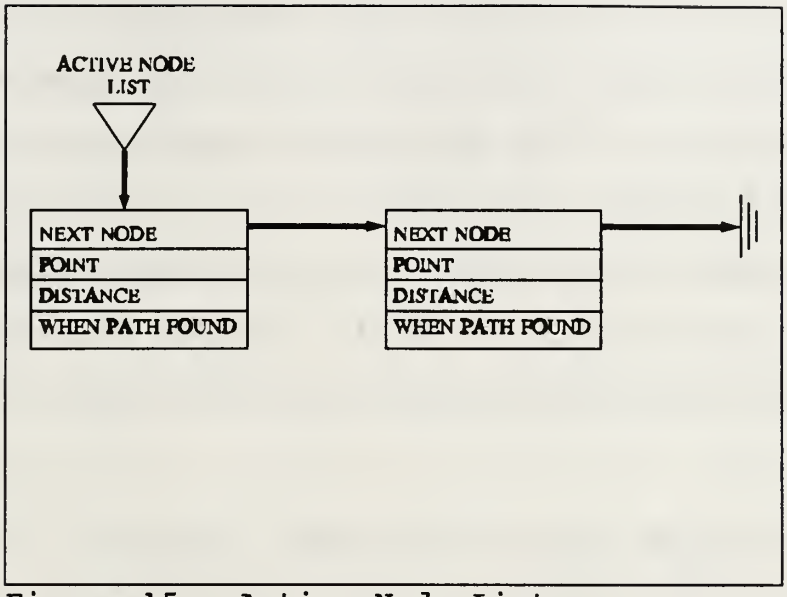

Figure 15 - Active Node List

This linked list is used in a heuristic which determines if a path being expanded from has been visited or not. If the point has already been visited, the distance entry in the structure is compared with the distance from the start to the last node in the expanded path plus the distance from the last node to the X Y Z point. If the distance in the structure of the link list is less than the computed distance, then the new point will not be added to the link list as an active node and the path will not extend to the this point. If the distance in the list is greater than the computed distance, then the new point is added to the link list as an active node and the path will be extended to the new point. In addition, the partial path with which the entry, when_path_found, equals the entry in the list, then the entry, able2expand, is marked as not

extendable. Also the distance and when_path_found entries are updated to reflect the data for this new added node to the partial path. If the node has not been visited in the path planning process, the node with its corresponding data is added to the active node list and the partial path is extended to that point. Table XIII depicts the pseudo language for this heuristic and the source code is in Appendix D.

Table XIII - CHECK ACTIVE NODE PSEUDO LANGUAGE

```
check_active_node_list(node_list, shortest_path, pt,
                       new_when_path_found)
{
  for(each active_node){
     if(active_node->pt == shortest_path->last_node){
       if(active_node->distance > shortest_path->cost +
                  distance(shortest_path->last_node, pt){
          mark_old_path_unexpandable(when_path_found);
          update_active_node(new_distance,
                             new_when_path_found);
          return(EXPAND);  /*EXPAND == 1*/
        }
        else  return(NOT_EXPANDABLE);
      }
  }
  /*node not visited yet*/
  insert_node_into_node_list(node_list, pt, cost);
  return(EXPAND);
}
```

8. Visibility

After the algorithm has found the shortest path to extend, the algorithm first determine if the last node added

to the path is visible either the goal or start point. This case is different from the visibility check used for the start and goal points because the last node can be on the backside of a polyhedron from the goal and still be visible with the goal as long as other polyhedrons lie between the last node and the goal. Figure 16 depicts this case. The concept of finding the path around the obstacle appears relatively simple; however the code is not. Table XIV lists the pseudo language I developed to solve this problem. Figure 17 identifies the points associated with both tangents of the situation shown in Figure 16.

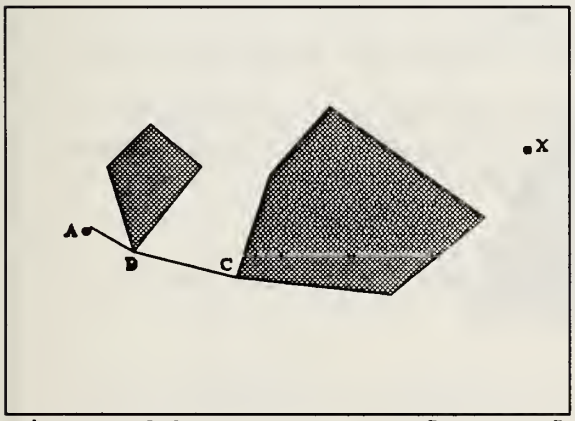

Figure 16 - Last Node Goal Visibility

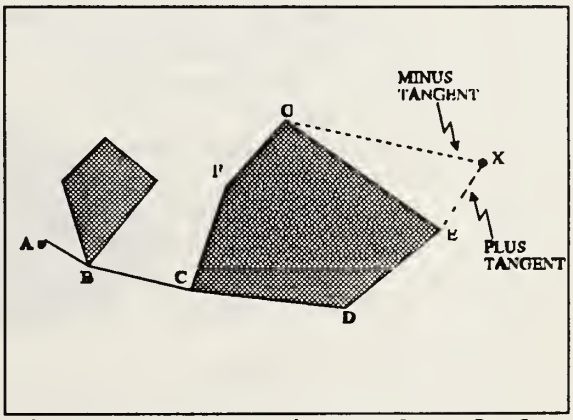

Figure 17 - Points found for both Tangents

Table XIV - PSEUDO LANGUAGE FOR VISIBILITY

```
Visibility(expand_point, goal_point, start, two_d_list,
           shortest_path)
{
  if(shortest_path forward)
     goal = start;
  else goal = goal_point;
  if(!intersect_polyhedron(expand_point, goal)){
     add_goal_to_goal_list;
     return(goal_list);
  if(shortest_path->type_tangent = VERTICAL)
     return(NULL);
  else{
    find_polygon_last_node_is_on;
    /*finds the plus tangent from goal to polygon
    for(each type of tangent from goal to the polygon)
      if(PLUS_TANGENT)
         from_polygon_pt =
            plus_tangent(polygon->polygon_start, goal);
       else
         from_polygon_pt =
          minus_tangent(polygon->polygon_start, goal);

    /*checks for intersection*/
       if(intersect_polyhedron(from_polygon_pt, goal))
         return(NULL);
       else{            /*two points are visible*/
         place goal and from polygon pt in goal list
         find from_polygon_pt on polygon
         if(PLUS_TANGENT){
           cycle counter-clockwise
           add points to goal list until arrive at
           expand_point
         }
         else{
           cycle clockwise, add points to goal list
           until arrive at expand point
         }
         if(direction_heuristic(shortest_path,
                                goal_list->pt)
            return(goal_list);
       }
    }
    deallocate and free memory of goal list
    return(NULL);
  }
}
```

In the pseudo language for the function visible, I refer to a function called "direction_heuristic". This function uses geometric concepts of polygons and rays to eliminate a path from extending incorrectly. If Figure 17 is used as an example, the "direction_heuristic" ensures that the nodes associated with the minus tangent are not chosen, while the nodes associated with the plus tangent. Figure 18 depicts why points F and G would not be chosen since point B can extend directly to G. Appendix F contains the source code for the "direction_heuristic" function.

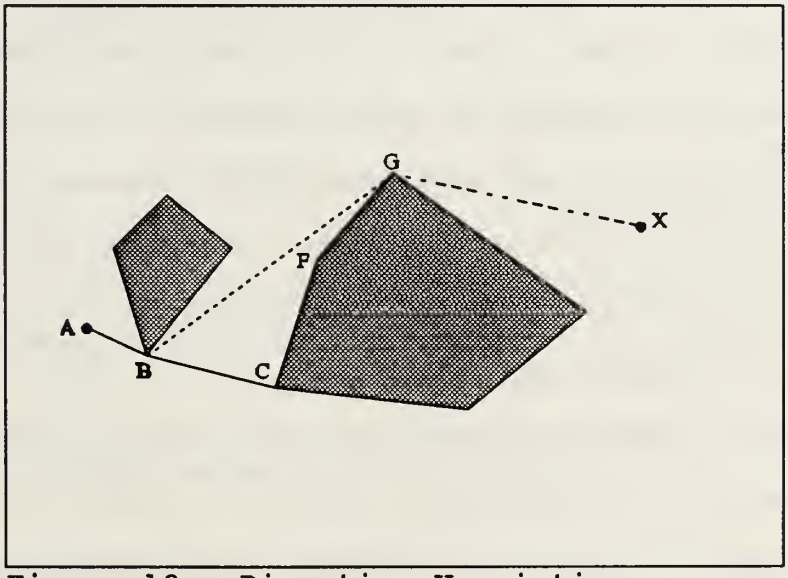

Figure 18 - Direction Heuristic Characteristics

The "visibility" function returns to the main function the goal list. If the goal list is not NULL, then the while loop is exited and the algorithm will add the goal list to the shortest path, update the header information of

the shortest path, delete all other paths, and terminate. If the goal list is NULL, then the algorithm enters the loop structure to begin extending the best path chosen to extend. Prior to extending the path, the algorithm prints to a file or to the screen the paths that are in the partial path list and the path chosen as the shortest path to extend.

B. THREE-DIMENSIONAL PATH PLANNING

As stated above, my three-dimensional path planning follows along the same concepts as Kanayama's two-dimensional algorithm. The basic algorithm begins by prompting the user to input the start and goal points. Once the user inputs these points, the algorithm checks whether or not there are obstacles lying between the two points. If there are one or more obstacle lying between the start point and goal, the algorithm then finds the two dimensional representation of the environment using the horizontal plane formed by the start and goal points as discussed earlier in this chapter. After finding this two-dimensional representation, the algorithm begins to find all the initial partial paths. After the algorithm finds these initial paths, it searches the path list to find the path to expand from after the initial paths are found. Next, the algorithm checks to determine if either the goal for a forward type path or the start point for a reverse path is visible from the last node added to the partial path. If it is, the goal

is added to that path, the path is marked as a completed path, all partial paths are deleted and associated memory deallocated, and the process ends. If the goal or start point is not visible, then the path is extended to all valid points in the environment and the process begins again with searching for a new path to expand. Table XV shows this basic algorithm in pseudo language.

Table XV - PATH PLANNER PSEUDO LANGUAGE

```
Three-dimensional_path_planner()
{
  Create_3D_world();
  Input_start_and_goal();
  if(!intersect_polyhedron(start, goal)){
     build_best_path(start_goal);
     return(best_path);
  }
  Two_D = build_two_dimensional_representation(start,
                                                goal);
  Partial_paths = Find_initial_paths(3D_world, Two_D,
                                     start, goal);

  while(visible(find_shortest_path(partial_paths),
                start_point, goal)){
  expand_shortest_path(shortest_path, 3D_world,
                       Two_D, goal,active_node_list);
  }
  Add_goal_and_mark_completed(shortest_path);
  Delete_partial_paths(partial_paths);
}
```

1. Initial Paths

In finding the initial paths, the main algorithm or the function "main(void) calls the functions "start_finding_paths". This function accepts the polygon

list, the start point and the goal point and returns to the main function the initial partial paths. Table XVI shows the pseudo language I developed for the function. Appendix F contains the source code for this function.

Table XVI - PSEUDO LANGUAGE FOR START FINDING PATHS

```
start_finding_paths(3d_world, polygons, start, goal)
{
  find_vertical_and_horizontal_planes
  polygons    =    build_2d_polygon_world(3d_world,
horizontal)
  for(each polygon in list){
    pt1 = plus_tangent(polygon_nodes, start);
    pt2 = minus_tangent(polygon_nodes, start);
    if(direction_heuristic_from_start(pt1, start, goal){
      if(visible(pt1, start)
        create_new_partial_path(partial_paths);
        add_start_goal(start, goal, new_path)
        update_header(new_path, goal);
      else
        partial_paths = up_and_over(start_point,
                                    3d_world, pt1);

    if(direction_heuristic_from_start(pt2, start, goal){
      if(visible(pt1, start)
        create_new_partial_path(partial_paths);
        add_start_goal(start, goal, new_path)
        update_header(new_path, goal);
      else
        partial_paths = up_and_over(start_point,
                                    3d_world, pt2);
  end for loop
  partial_paths = up_and_over(start, goal, 3d_world);
```

The function "start_finding_paths" begins by finding the vertical and horizontal planes formed by the start and goal. It then proceeds to find the polygons formed by the intersection of this horizontal plane and the polyhedrons in the world. After finding this polygon list, the process starts with the first polygon in the list and finds both the

plus and minus tangent points with respect to the start point as described earlier in this chapter. The process then uses a heuristic to determine if the direction of the ray formed by the start and goal is within plus or minus 180 degrees of the ray formed by the start and goal points. If this heuristic determines that the ray is valid, then the process continues by making additional checks. If the heuristic determines that the ray is not valid, the process stops checking the plus tangent point and begins the process over checking the minus tangent point.

Once the process determines if the ray is valid, it begins to check if the start point is visible with this plus tangent point. The process uses the polygon list to determine if the two points are visible. If a two points are not visible using this two-dimensional polygon list, then the two points will not be visible using polyhedron list since the polygon list is a subset of the polyhedron world. By using this smaller polygon list, the process can check for visibility faster than if it used the polyhedron list.

In determining that the two points are visible, the process begins to find a partial path which corresponds to the type of path in which all nodes are on the same plane. If the two points are visible then the process adds a new partial path to the partial path list. This partial path list has the start point as the first point and the plus

tangent point as the last_node. In addition, the process also computes the header data of the partial path.

If the process determines that the two points are not visible, then the process begins the phase of finding the type of path which goes over one or more polyhedrons and then around (at least one) the rest of the polyhedrons to the goal.

In finding the this path, the process calls the function "up_and_over", passing to the function the partial paths pointer, expansion point, start point and the either the plus or minus tangent point, depending on which point is being checked. The function returns the partial path list adding to the list the vertical path as described earlier in this chapter. The function up_and_over and the functions this function calls can be found in Appendix J for further analysis.

After each polygon in the two-dimensional list has been checked, the "start_finding_function" then finds the partial path which travels over all the polyhedrons lying between the start and goal. It does so by using the same function used above. In using the "up_and_over" function, the calling function passes the same parameters, except the goal point is passed instead one of the two tangent points.

The last step the process determines is to find the paths which go around one or more polyhedrons and then over (at least one) the remaining polyhedrons. This step is not

as difficult as it appears. Upon analyzing this step, I noticed that this step was the similar to the step of finding all the paths which go over one or more polyhedrons and around the remaining only in reverse order. As a result, to find these type of paths, I developed a sub-process which extends from the goal and not the start point.

The "start_finding_paths" function calls the function "up_and_over_from_goal". I developed this function along the same lines as "start_finding_paths". For each polygon in the two-dimensional list, the sub-process finds both tangents points with respect to the goal, the expansion point. After finding the these points, the sub-process first determines if the expansion point is visible to the plus tangent point. If the points are not visible, the "up_and_over" function is called, adding paths extended from the goal. After checking the plus tangent point, the sub-process then performs the same operations for the minus tangent point. To distinguish these paths from the paths extended from the start point, the "forward" integer flag in the header information of the path is flagged equal to 0 while this flag is equal to 1 if the path extends from the start. Upon returning control back to "start_finding_paths", the function returns the partial path list, which in turn returns the same list back to the main function. As with the other vertical function, the "up_and_over_from_goal" function is listed in Appendix J.

When control of the process returns to the main function, all the partial paths which are needed to find the best path are included in the list. From this list, the process searches for the best path from which to extend from.

2. Extending Partial Paths

In Extending a partial path, the algorithm enters a while loop structure which first searches the partial path list to find the best path to extend, secondly determines the visibility of the last node of the best path chosen, and thirdly, based on the results of checking the visibility, extends this path or exits the loop.

a. Search Technique

After finding these initial paths, the process uses an A* (A Star) search to determine which path to extend. When considering the various paths for expansion, the A* search uses the function f(n) = g(n) + h(n) where g(n) is the actual cost of the path from either the start if a forward path or the goal if a reverse path; h(n) is the heuristic estimate or the cost/distance from either the last node to the goal if the path is a forward path or the last node to the start point if the path is a reverse path. In the header data of each path g(n) is the "cost" entry; h(n) is the "estimated cost" entry; and f(n) is the "total cost" entry. The minimum f(n) of all possible paths identifies

the shortest partial path and determines which path the algorithm will use to extend to the next set of nodes.

The advantage of using the A* Search is that the heuristic measure, h(n) is a lower bound of the actual cost of the shortest path. As a result, the algorithm is admissible, finding an optimal path if one exists. Once the A* Search has determined which path to extend, the algorithm begins a process similar to the one used to find the initial paths.

b. Extending the Shortest Path

Extending the shortest path begins with the function "extend_path". The main function passes to "extend_path" the following parameters: start point; goal; and pointers to the shortest path, polygon list, polyhedron list and the active node list. It returns to the main function the pointer to the shortest path.

Table XVII lists the pseudo language I developed to extend the path. As can be seen, the concept behind the function is to check the type of tangent entry in the header information of the shortest path. If the type tangent is not a vertical tangent, then the path is extended on the plane slice indicated by the plane number entry in the header information. However, if the type_tangent entry is vertical, then a new plane slice must be used. In addition, the function is Appendix F.

Table XVII - PSEUDO LANGUAGE FOR EXTEND_PATH

```
extend_path
{
  if(shortest_path->type_tangent != VERTICAL)
    find_tangents_from_inner_nodes;
  else{
    add_on_new_plane_intersection;
    continue_vertical_path;
  }
  return(shortest_path);
}
```

c. Vertical Extension

In extending the path in which the tangent to the last node of the path is a vertical tangent, the process first finds the horizontal plane formed by the last node of the path and the goal. Then the process finds the intersection of the plane with each polyhedron in the world. These new polygons are appended to the list of the existing polygons with the plane_number entry corresponding to the number of horizontal planes used finding polygon intersections.

After finding the polygon intersections for the new horizontal plane slice, the process must accomplish two tasks. The first task is the find the type of paths which correspond to the last two types of paths as referred to earlier in the chapter. The second task is to extend the path to the one node on the horizontal plane slice found directly before beginning this process.

To find the paths which begin traversing over one or more obstacles, then around one or more obstacles and then over at least one obstacle or similarly around, over and then around are found from a vertical point. As with the paths which correspond to the third and fourth types of depending on the direction of the path. If the direction of the path is forward, then the type of path found is up, around and up; however, if the direction is from the goal, the type of path is around, up, and around. Since finding these two types of paths are the hardest to find, I left them for the last task to accomplish and at the time of writing of this paper I have not implemented the process into the algorithm which I developed. However, Table XVIII shows the process written in pseudo language.

Table XVIII - PSEUDO LANGUAGE FOR FUNCTION UP, OVER, UP

```
up_over_up(path)
{
  For(each polyhedron){
    from = find top face(1st_polyhedron);
    for(each polyhedron not the 1st polyhedron){
      to = find top face (2nd_polyhedron);
      for(loop 4 times) /*there are 4 tangents to
                         find*/
        tangent_list = tangent(to, from, mode1, mode2);
        if(check_tangent_list(tangent_list){
           free_tangent_list(tangent_list)
           continue;
        }
        vertical_list = find_vertical_list(tangent_pt1,

tangent_pt2);
        last_node = find last node in vertical_list;
        new_poly_list = build two-d polygon list
                              (last_node, path's
                                    last_node)
        new_path_list = duplicate_path(path)
        new_path_list = continue_vertical_path using
                      new_poly_list
        while(not(visible(
                find_shortest_path(new_paths_list),
                new_poly_list, tangent_Pt1)
           new_path_list = extend_path(new_path_list,
                          tangent_pt1,new_poly_list);
            /*tangent_pt1 is new goal point*/
        }
        add tangent_pt1 to shortest path;
        using this process' shortest path
          if(direction heuristic with last node in path
             and last node in vertical_path)
            add vertical list to shortest path
          else mark path as not able2expand
          delete all partial paths in new list except
          shortest
          place in locally shortest_path_list
          free_new_poly_list(new_poly_list);
    }
  }
  add locally shortest path list to partial path list
    after passed in path and connect links
  return(path);
}
```

In beginning the second task of the process, the process uses these new polygons and finds the one node which extends the path along the same ray direction in the header information. In doing so, the process scans these new polygons checking the polyhedron entry. If this entry equals the entry of polyhed4vert, then the process finds the two horizontal tangents from the last node to the polygon. The process then determines the rays of these two tangents. The process adds to the path as the last node the tangent point whose tangent direction is in the same general direction as the ray_direction entry in the header information.

In determining if the two rays are in the same general direction, I had to consider the case where the first plane slice created a polygon with x and y entries in the link list that are different from the x and y entries in the link list of the polygon formed by the new plane slice. Since the first plane and its associated polygons were used to determine the vertical points of the path, the new polygon used to determine the tangent point will result in a different path if this tangent point and start point are used.

In determining if the point should be added, I modified the process to first check the type_tangent entry in the structure which the pointer polyhed4vert_PTR is linked to. The process then executes the correct tangent

function (either plus_tangent or minus_tangent). The next check the process does is to determine if the difference in ray directions is less than .0001. Instead of using zero, I use this value since this algorithm may be used on different platforms and the precision of the platforms may be different. If the difference is, then the point is added to the list. If the difference is not, then the process must recalculate the vertical intersection points using the vertical plane formed by the start point and this new point. In both cases, the header information is updated accordingly and the function returns to calling function, extend_path, the shortest path. The function "continue_vertical_path" is in Appendix J.

(1) Horizontal Extension

If the function "extend_path" determines that the shortest_path's type_tangent entry does not indicate a vertical tangent, then the process begins to find all valid tangents on the plane slice indicated by the plane_number entry in the header information. The function which performs this work is "find_tangents_from_inner_nodes" and the source code is in Appendix F.

The main concept behind this function is to cycle through the polygon list. For those polygons which have the same plane number as the shortest path's plane_number entry, the tangent formed by the polygon being

checked and the polygon which the last node of the shortest path lies as described earlier in this chapter.

The process continues on to check whether the ray formed by the two tangent points intersects any of the polyhedrons in the world. If there is not, then the shortest path is duplicated entirely, the two tangent points are added to the duplicated path. In addition, all points which lie on the same polygon as, are between the shortest path's last node and one of the tangent points, and are in the direction (either plus or minus) the path is advancing are added to the path between these two points. After completing the duplication and insertion of the new points, the header information is updated and the process inserts the new partial path into the partial path list after the shortest path, reconnecting the link list accordingly.

C. FINDINGS AND EXAMPLES

As I have already indicated, the number of paths grows in polynomial time and is dependent not only on the number of polyhedrons but also the number of faces in each polyhedron and number of nodes in each face. As a result, I will limit the number of polyhedrons in the environment to 1 and 2. In addition, I have included a print function which prints out each partial path listing prior to an iteration of extending the shortest path. A listing of for each example is in Appendix M.

1. One Polyhedron World

For the 1 polyhedron world, I use the polyhedron described in Table XIX. In the first example, I use the starting point (1, 15, 14) and the goal point (40, 15, 14). I use the same depth value (z value) in the example to show that the plane slice does correctly work. Figure 19 depicts the scenario with a top down view for this example.

In generating the initial partial path list, the algorithm calculated 3 partial paths. Using the same top down view of Figure 19, Figure 20 shows the partial paths with nodes of the path labeled.

The reader should notice that the nodes of the path which end at a vertex of the polygon in Figure 20 are nodes of the polygon found using the horizontal plane formed by the start and goal points. These two paths also represent the type of paths in which all nodes of the path lie on the same plane. Similarly, the one path in Figure 20 which does not have a tangent to the polygon as a node of the path represents the path which traverses over all polyhedrons to the goal. In addition, the intersection points in Figure 20 represent the intersection of the vertical plane and the top face of the polyhedron.

Table XIX - ONE POLYHEDRON WORLD

SIDE FACE 1: (10 7 15) (10 20 15) (10 20 3) (10 7 3)
SIDE FACE 2: (10 20 15) (14 20 15) (14 20 3) (10 20 3)
SIDE FACE 3: (14 20 15) (14 7 15) (14 7 3) (14 20 3)
SIDE FACE 4: (14 7 15) (10 7 15) (10 7 3) (14 7 3)
TOP FACE: (10 7 3) (10 20 3) (14 20 3) (14 7 3)
BOTTOM FACE: (10 7 15) (10 20 15) (14 20 15) (14 7 15)

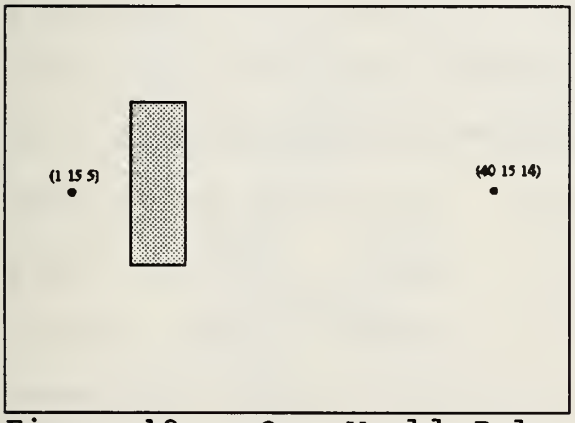

Figure 19 - One World Polyhedron (Top Down View)

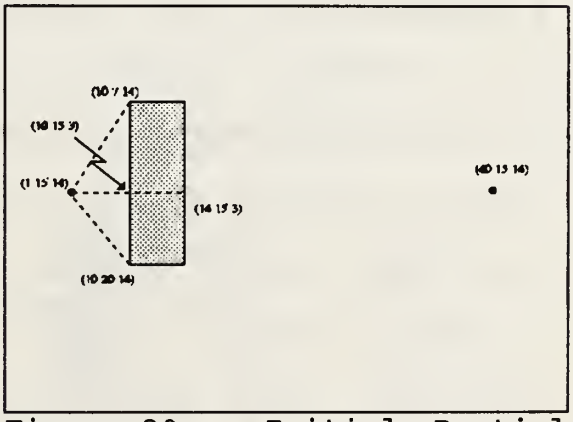

Figure 20 - Initial Partial Paths

The algorithm chose the path (1 15 14) to (10 20 14) to extend using the A* search. In checking the visibility of the point with the goal, the algorithm determined that the expansion point was visible. As a result the correct node(s) of the polygon were added to the path along with the goal, the header information of the path updated, all other

paths, the polygon list and the polyhedron list are deleted from memory and the algorithm terminates with the best path as (1 15 14) to (7 20 14) to (14 20 14) to (40 15 14).

In the second example using the one polyhedron world, I used the points (1 13 5) and (1 40 14) for the start and goal points respectively. From the top down view, these initial partial paths look similar to the paths in the previous example; however, since the start and goal points are not at the same depth, the intersection of the polyhedron and the horizontal plane produces a polygon with different depth values than the first example as shown in Figure 21.

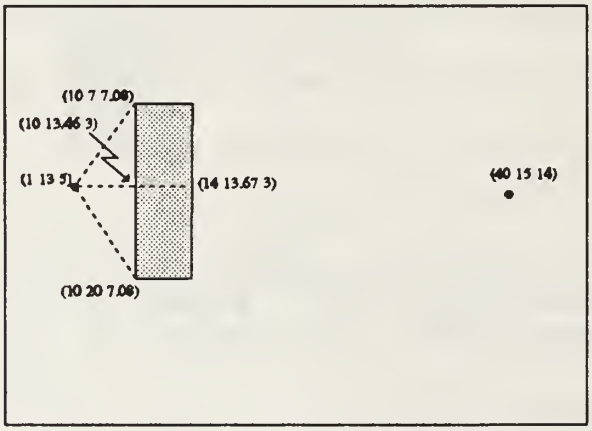

Figure 21 - Second Example for 1 Polyhedron

In this example, as expected, the shortest path is the vertical path. Since the vertical path is directly visible to the goal, the algorithm adds the goal to the path list and again, it performs all the administrative functions

to update the header information, deletes all appropriate paths and terminates.

2. Two Polyhedron World

For the 2 polyhedron world, I use the same polyhedron as in the examples for a one world polyhedron. Table XX lists the coordinates of the nodes of the second polyhedron. In addition, I use the same start and goal points in both examples, similar to the first two examples. Figure 22 shows the initial partial paths. As can be seen and expected, the number of paths has increased with the process including of the vertical paths over the first polyhedron. In addition, since the second polyhedron is taller than the first polyhedron (z values are smaller, thus at a shallower depth), Figure 22 also shows those nodes of a path marked with an "x". With respect to the one vertical path from start to goal, close observation will show that the one node on the edge of the top face closest to the second polyhedron was not included in the path. This node is not needed since the nodes in between this node are visible to each other. As in the first example, the process chose the partial path (1 15 14) to (10 20 14) as the path to extend. Figure 23 shows only those partial paths which are generated from extending the shortest path. Since there are obstacles between the last node of the path and the goal, the process chooses the partial path (1 15 14) to (10

20 14) to (14 20 14) to (20 20 14) for the next iteration of extending the shortest path. This partial path is visible to the goal so the algorithm as before, performs the administrative functions and terminates.

Table XX - SECOND POLYHEDRAL LISTING

(20 7 15) (20 20 15) (20 20 1) (20 7 1)
(20 20 15) (24 20 15) (24 20 1) (20 20 1)
(24 20 15) (24 7 15) (24 7 1) (24 20 1)
(24 7 15) (20 7 15) (20 7 1) (24 7 1)
(20 7 1) (20 20 1) (24 20 1) (24 7 1)
(20 7 15) (20 20 15) (24 20 15) (24 7 15)

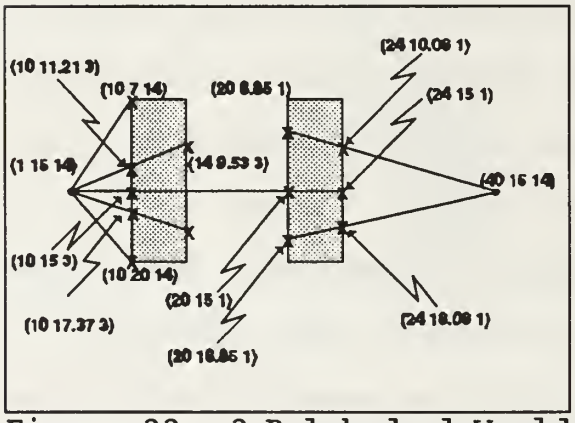

Figure 22 - 2 Polyhedral World and Initial Partial Paths

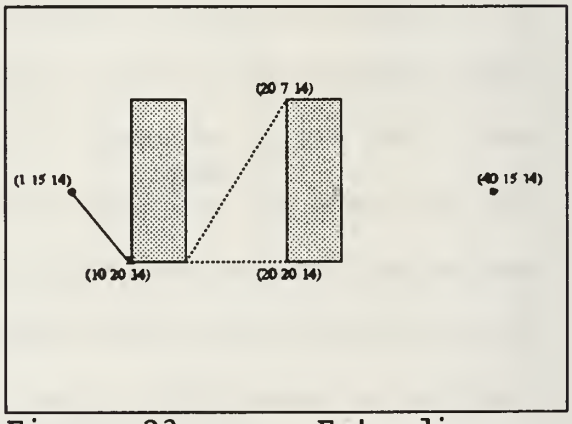

Figure 23 - Extending a Shortest Path in a Two Polyhedral World

The second example using a two polyhedral world has the points (7 13 7) and (40 15 14) as the start and goal points. Figure 24 shows those paths generated from the start. As in the last example, the second polyhedral

obstacle is taller than the other two polyhedral obstacles. As a result, the vertical paths which extend to this obstacle will have only one intersecting point on the first obstacle the path intersects. Again, the nodes of the path are displayed with an "x" at the intersection point. After developing these initial paths, the algorithm iterates through the process 4 times before the expansion point is visible with the goal. Figures 25, 26, 27 and 28 depict the process the algorithm used to find the goal.

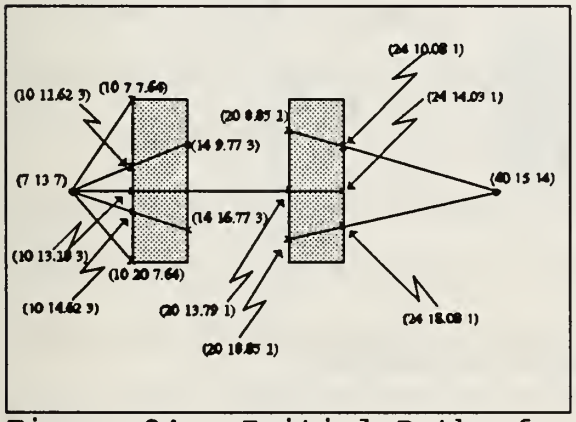

Figure 24 - Initial Paths for Two Polyhedral World (Example 2)

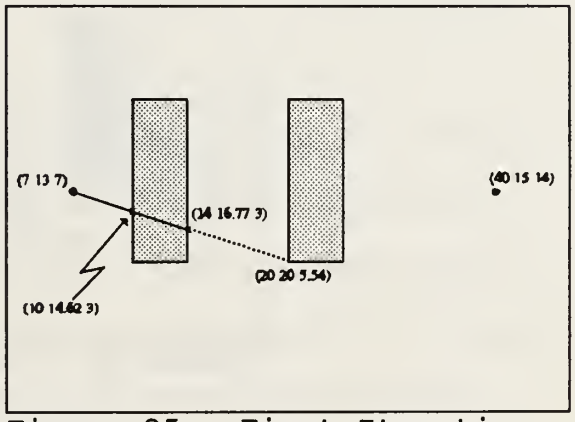

Figure 25 - First Iteration

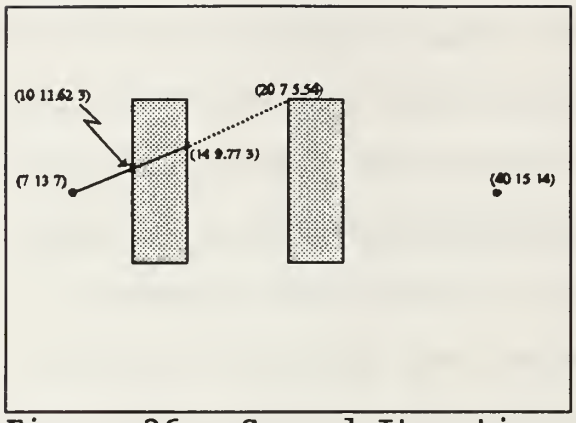

Figure 26 - Second Iteration

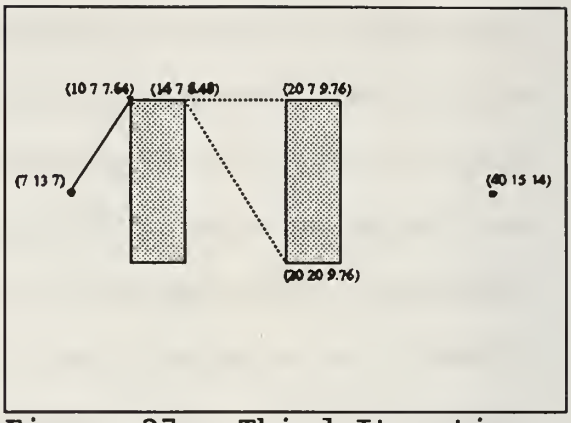

Figure 27 - Third Iteration

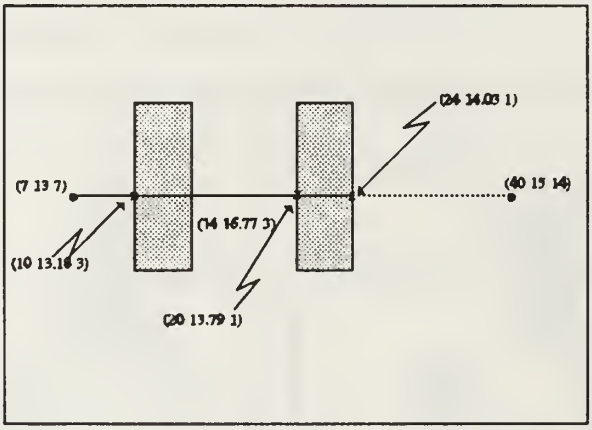

Figure 28 - Fourth Iteration

V. RECOMMENDATIONS AND CONCLUSIONS

The algorithm in which the source code is based on does not end up finding the shortest path from a start to goal, but a best path for the two points for a point traversing the environment. The concept of three-dimensional path planning was considerably more difficult then I imagined. As a result, I was not able to implement in the source code the portion of the algorithm which solves the last two types of paths presented in Chapter IV. However, if implemented, the shortest path can be found.

I initially started work on solving/finding the type of path which lies on the same plane as the start and goal points. In solving each of the other 3 remaining types of paths which this thesis addresses, I found myself developing again how I solved the previously solved types of paths. Some changes were as simple as changing the number of parameters passed into the function or the structure used in passing the return value back from the called function to the calling function. Other changes included major modifications.

On area which of source code which took many iterations and time in forethought was the header information structure for each path. Every time I proceeded in solving the next type of path, I discovered I needed more information or in

the case of changing a function's structure not needing specific datum.

This algorithm is only the first step in implementing a path planner in the NPS II AUV. In addition to including in the source code the portion of the algorithm which solves the last two types of paths, other modules will need to be added.

One of the most important to me added is a module which insures that the path being extended one more node is a safe path for an object larger than a point traversing the world. Since the NPS II AUV can be thought of as a cylinder traveling through the world, the concept of safe path planning is not finding the tangential lines to the polyhedrons but finding a cylinder which intersects the polyhedron at one point. This concept is not as easy as it appears. One particular difficult problem will be in dealing with polyhedrons in which the sides are not perpendicular to the ground.

Other modules which should be added after the safe path module is added concern implementing the coefficient characteristics of the AUV in the path planner as well as sea and weather conditions and procedures which will aggressively limit the size of the world. The order of magnitude for this path planner is driven by not only the number of polyhedrons, but also the number of nodes in each polyhedron. By reducing the search space, time can be saved

in the planning process. With regards to the conditions and coefficients, they not only affect the performance of the AUV, but they also affect the determination of a best path and need to be included.

LIST OF REFERENCES

Blidberg, D., Chappell, S., Jalbert, J., Turner, R., Sedor, G., Eaton, P., "The EAVE AUV Program at the Marine Systems Engineering Laboratory," *The 1st Workshop on: Mobile Robots for Subsea Environments, International Advanced Robotics Programme*, pp. 33-42, 1990.

Floyd, C., *Design and Implementation of a Collision Avoidance System For the NPS Autonomous Underwater Vehicle (AUV II) Utilizing Ultrasonic Sensors*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

Good, M., *Design and Construction of a Second Generation Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

Herman, M., "Fast Three-Dimensional, Collision_Free Motion Planning," *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pp. 316-321.

Jurewicz, T., *A Real Time Autonomous Underwater Vehicle Dynamic Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

Kanayama, Y., *Two Dimensional Spatial Reasoning (Spatial Planning?)*, Class Notes, CS4313, Naval Postgraduate School, Fall 1990.

Magrino, C., *Three Dimensional Guidance For The NPS Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

Ong, S., *A Mission Planning Expert System with Three Dimensional Path Optimization for the NPS Model 2 Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1990.

Pappas, G., Shotts, W., O'Brien, M., and Wyman, W., "The DARPA/Navy Unmanned Undersea Vehicle Program," *Unmanned Systems*, pp. 24-30, Spring 1991.

Robertson, R.C. "National Defence Applications of Autonomous Underwater Vehicles", *IEEE Journal of Ocean Engineering*, vol OE-11, No. 4., pp. 462-467, Oct 1986.

Schartz, J., Sharir, M., "A Survey of Motion Planning and Related Geometric Algorithms," *Artificial Intelligence*, vol 37, no 1-3, pp. 157-169, December 1988.

Sharir, M., "Efficient Algorithms for Planning Purely Translational Collision-free Motion in Two and Three Dimensions," *Proceedings of the 1987 IEEE international Conference on Robotics and Automation*, vol 3, pp. 1326-1331.

Texas A&M University, Information Brief Packet of the AUVC Project, 1 November 1990.

Ura, T., "Development of AUV PTEROA," *The 1st Workshop on" Mobile Robots for Subsea Environments, International Advanced Robotics Programme*, pp. 195-200, 1990.

Warren, C., "A Technique For Autonomous Underwater Vehicle Route Planning," *IEEE Journal of Oceanic Engineering*, vol 15, no. 3, pp. 199-204, 3 July 1990.

Warren, C., "Global Path Planning using Artificial Potentials Fields," *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pp. 316-321.

Wilkinson, P. *A Mission Executor For an Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

Zyda, M., McGhee, R., Kwak, S., Nordman, D., Rogers, R., and Marco, D., "Three-Dimensional Visualization of Mission Planning and Control For The NPS Autonomous Underwater Vehicle," *IEEE Journal of Oceanic Engineering*, vol 15, no. 3, pp. 217-221, 3 July 1990.

APPENDIX A

This appendix contains the source code which is in the file main.c

```
#define MAIN

#include "3d_tan.h"
#include "plot.h"

/*-------------------------------------------------------------------------
                                    MAIN
-------------------------------------o0o-------------------------------------*/
void            main()
{

   int              choice, iteration_count = 0;
   equation         plane_equation;
   paths           *current_path = NULL;
   a_path          *goal_path = NULL;;

   decision();

   polyhedron_list = create_obstacle_list();

   boundry = assign_boundry_points();
   printf("\nSTART POINT");
   start_point = input_point(start_point);
   printf("\nGOAL");
   goal = input_point(goal);


   inside_boundries(choice, boundry, start_point, goal);
   goal_direction = atan2(goal.y - start_point.y,
                     goal.x - start_point.x);


   /* CHECK VISIBILITY BETWEEN START AND GOAL */
   if (!intersect_polyhedron(start_point, goal)) {
      printf("The best path to follow is from the start to goal");
      exit(1);

   }
   /* FIND PARTIAL PATHS FROM THE START TO 1ST SET OF TANGENTS */
   start_path = start_finding_paths(start_path, start_2d, start_point, goal,
                              boundry, head_active_nodes);
   print_each_path(start_path, ++iteration_count);

#ifdef DOS
   Pause();
#endif

   /* FINDS THE BEST PATH FROM THE INITIAL PARTIAL PATH IN THE PATH LIST */

   while ((goal_path = visibility(find_shortest_path(start_path),
                                  goal, start_point, start_2d)) == NULL) {
      if(1 != iteration_count)
       print_each_path(start_path, iteration_count++);
      else
       iteration_count++;
      print_path_shortest(shortest_path);
      shortest_path = extend_path(start_point, goal, shortest_path, start_2d,
                           boundry, head_active_nodes);
   }
   fprintf(fpt2, "\n\nTHE GOAL HAS BEEN REACHED");
```

```
   print_each_path(start_path, ++iteration_count);
   print_path_shortest(shortest_path);
   shortest_path = goal_reached_update(goal, shortest_path, goal_path);

#ifdef DOS
   if (plot2screen)
      draw_shortest_path(shortest_path);
#endif

   print_each_path(start_path, ++iteration_count);
   print_path_shortest(shortest_path);

#ifdef DOS
   if (plot2screen) {
       closegraph();
   }
#endif

   fclose(fpt2);
}

/*-------------------------------------------------------------------------
                                    INPUT POINT
---------------------------------------o0o-------------------------------
This function allows the user to input a pt into a structure (record)
and returns that structure to the calling function
*/

point
input_point(pt)
   point           pt;
{
   double          x_coord, y_coord, z_coord;

   printf("\nInput x, y, z coordinate:  ");
   scanf("%lf %lf %lf", POINT_VALUES);  /* macro to assign values to x,y,z */
   pt.x = x_coord, pt.y = y_coord, pt.z = z_coord;
   return (pt);
}


/*-------------------------------------------------------------------------
                                     DECISION
---------------------------------------o0o-------------------------------
*/
int             decision()
{
   int             choice = 0;
   char            answer;

   do{
      printf("\nCHOOSE ONE\n1.  IN POOL 1\n2.  IN POOL 2\n3.  IN POOL 3\n4.  IN POOL
4\n");
      printf("5.  IN POOL 5\n");
     scanf("%d", &choice);
   /* ESTABLISH THE ENVIRONMENT */

     switch(choice){
       case 1:
         filename = "pool.dat";
         break;
       case 2:
         filename = "pool1.dat";
        break;
       case 3:
         filename = "pool3.dat";
        break;
       case 4:
         filename = "pool4.dat";
```

```
        break;
      case 5:
        filename = "pool2.dat";
        break;
      default:
        printf("\n\nINCORRECT ENTRY.  Try again");
        clrscr();
      }
   }while(!choice);

   printf("Do you want to print results to a file (y or n)? ");
   fflush(stdin);
   answer = getc(stdin);

   if (answer == 'y' || answer == 'Y') {
      print2file = 1;
      printf("\nThe file name is path.doc.\n");
   }
   else
      print2file = 0;
   if (print2file) {
      if ((fpt2 = fopen("path.doc", WRITEONLY)) == NULL) {
       perror("  Error:  Data file for wrinting did not open correctly.\n ");
       exit(0);
      }
   }

#ifdef DOS
   printf("\nDo you want a plot of this path on the screen (y or n)?");
   fflush(stdin);
   answer =  getc(stdin);
   if (answer == 'y' || answer == 'Y')
      plot2screen = 1;
   else
      plot2screen = 0;
#endif

   printf("\nDo you want a horizontal path only (y or n)?  ");
   fflush(stdin);
   answer = getc(stdin);
   if('y' == answer)
      horizontal = 1;
   else
      horizontal = 0;

}
```

```
/*------------------------------------------------------------------------
                                    INSIDE BOUNDRY
----------------------------------------o0o-------------------------------
This function checks whether or not the start point and goal point/points
are within the boundries of the pool.  It calls boundary check passing to
it the point to be checked.*/

void            inside_boundries(choice, boundry, start_point, goal)
   int             choice;
   boundries       boundry;
   point           start_point, goal;

{
   int             boolean = FALSE;

   if (choice && (!boundry_check(boundry, start_point))) {
      printf("\nThe start point is not within the boundaries\n");
      boolean = TRUE;
   }
   if (choice && (!boundry_check(boundry, goal))) {
      printf("\nThe goal is not within the boundaries\n");
      boolean = TRUE;
   }
   if (boolean)
      exit(1);

}

/*------------------------------------------------------------------------
                                    BOUNDRY CHECK
----------------------------------------o0o-------------------------------
This function determines if the passed in point lies within the boundries
of the search space.  If it is not, a FALSE (0) is return, else a TRUE (1)
is returned to the calling function.*/

int             boundry_check(boundry, pt)
   boundries       boundry;
   point           pt;
{
   if (pt.x > boundry.x1 && pt.x < boundry.x2 &&
       pt.y > boundry.y1 && pt.y < boundry.y2 &&
       pt.z >= boundry.z1 && pt.z < boundry.z2)
      return (TRUE);
   else
      return (FALSE);
}
```

```
/*-------------------------------------------------------------------------
                                ASSIGN BOUNDRY POINTS
-------------------------------------o0o-----------------------------------
This function assigns the boundry points to the search area.  At present
it is fixed at the pool dimensions used in the pool.dat; however, at a
later date, input function can be encorporated to allow the user to speci-
fy the search area.
*/

boundries
assign_boundry_points()
{
   boundries       boundry;

   boundry.x1 = 0;
   boundry.x2 = 48;
   boundry.y1 = 0;
   boundry.y2 = 34;
   boundry.z1 = 0;
   boundry.z2 = 15;

   return (boundry);
}


/*-------------------------------------------------------------------------
                                FIND SHORTEST PATH
-------------------------------------o0o-----------------------------------
This function finds the shortest path and assigns the global pointer to
it*/

point
find_shortest_path(start_path)
  paths *start_path;
{
   /* next for structure is used only to find shortest path */
   paths          *current_path;
   point           pt;

   shortest_path = NULL;

   for (current_path = start_path, shortest_path = current_path;
       current_path != NULL;
       current_path = current_path->next_path) {

        if ((current_path->total_cost < shortest_path->total_cost)
             && current_path->able2expand)
           shortest_path = current_path;
        if(!shortest_path->able2expand)
           shortest_path = shortest_path->next_path;
   }
   if (plot2screen)
      id_shortest_path_for_expansion(shortest_path);

   return (shortest_path->last_node->pt);
}
```

APPENDIX B

```
#include "3d_tan.h"

/*-------------------------------------------------------------------------------
                              BUILD TWO D POLYGON LIST
---------------------------------------o0o-----------------------------------------
This function is the major function which builds the two dimensional environment
from the 3 dimensional environment.  It accepts as a parameterthe perpendicular
plane (labeled plane) and places into a link list theintersection points of a
polyhedron with the plane.  These intersectionpoints form the polygon.
*/

polygon_list   *build_two_d_polygon_list(start_2d, plane, null_ptr)

   polygon_list   *start_2d;
   equation        plane;
   int             null_ptr;
{
   obstacle_list              *current_polyhedron = polyhedron_list;
   polyhedron_obstacle        *current_face;
   polyhedron_obstacle_plane *current_node;
   polygon_list   *polygon, *last_polygon, *previous;
   int             count, intersection_count, boolean = FALSE;
   static int      polygon_number = 1;
   point           pt1, pt2;

   plane_number++;
   if(start_2d != NULL){
     for(last_polygon = start_2d;                /*finds last 2d polygon*/
      last_polygon->next_polygon != NULL;
      previous = last_polygon, last_polygon = last_polygon->next_polygon);
   }
   for (current_polyhedron = polyhedron_list; current_polyhedron != NULL;
         current_polyhedron = current_polyhedron->next_polygon) {

      if (start_2d == NULL) {
         NEW_POLYGON(start_2d);
         last_polygon = start_2d;
         plane_number = 1;
      }
      else {
         if (null_ptr) { /* create a new node if the last node is not null */
           NEW_POLYGON(last_polygon->next_polygon);
           previous = last_polygon;
           last_polygon = last_polygon->next_polygon;
         }
      }
       last_polygon->polygon_start = NULL;
       last_polygon->next_polygon = NULL;
       last_polygon->plane_number = plane_number;
       last_polygon->polygon_number = polygon_number++;
       last_polygon->polyhedron_number = current_polyhedron->polyhedron_number;

      for (current_face = current_polyhedron->obstacle, intersection_count = 0;
            current_face != NULL;
            current_face = current_face->next_face) {

          for (current_node = current_face->face_nodes,  count = 1,
               boolean = FALSE;
              count <= current_face->face_nodes->number_nodes;
```

```
                current_node = current_node->ccw, count++) {

              if (boolean == FALSE) {
                ptl = intersection_point(current_node->pt, current_node->ccw->pt,
                                         plane);
                if (ptl.x != NO_INTERSECTION) {
                   boolean = TRUE;
                   intersection_count++;
                   break;
                }
              }
            }
            if (boolean) {
              last_polygon->polygon_start = create_list(last_polygon->polygon_start,
                                                        intersection_count, ptl);
            }
          }
          if (last_polygon->polygon_start != NULL) {
            null_ptr = TRUE;
            last_polygon->polygon_start->number_nodes =
                              count_polygon_nodes(last_polygon);
             last_polygon = remove_duplicate_nodes(last_polygon);
             last_polygon = connect_links(last_polygon);
            }
            else
            null_ptr = FALSE;

        }
        if(last_polygon->polygon_start == NULL)
          previous->next_polygon = NULL;
        return (start_2d);
}

/*-------------------------------------------------------------------------------
                                    CREATE LIST
---------------------------------------o0o-------------------------------------
This function accepts the polygon_start, the intersection_count, and the ptl
parameters and places the point (ptl) into the link list for 2d polygons
(polygon_start).
*/

obstacle_plane *create_list(polygon_start, intersection_count, ptl)
   obstacle_plane *polygon_start;
   int             intersection_count;
   point           ptl;
{
   obstacle_plane *temp_ptr, *current, *previous;
   int             count;
   obstacle_plane *two_d_ptr;

   if (polygon_start == NULL) {
      NEW_PLANE(polygon_start);
      polygon_start->pt = assign_point_values(ptl);
      polygon_start->number_nodes = intersection_count;
      polygon_start->ccw = polygon_start->cw = NULL;
   }
   else {
      NEW_PLANE(temp_ptr);
      temp_ptr->pt = assign_point_values(ptl);
      temp_ptr->ccw = NULL;

      for (previous = polygon_start, current = polygon_start, count = 1;
         current != NULL;
         previous = current, current = current->ccw, count++) {

        if (current->pt.x == ptl.x && current->pt.y == ptl.y &&
            current->pt.z == ptl.z) {
          intersection_count -= 1;
```

```
            polygon_start->number_nodes = intersection_count;
            free(temp_ptr);
            return (polygon_start);
          }
        }
        previous->ccw = temp_ptr;
       polygon_start->number_nodes = intersection_count;
    }
    return (polygon_start);
}

/*-------------------------------------------------------------------------------
                                ASSIGN POINT VALUES
---------------------------------------o0o-----------------------------------------
This function assigns the coordinate values (x, y, z) to ptl which is declared as
a point structure
*/

point           assign_point_values(pt2)
    point           pt2;
{
    point           ptl;

    ptl.x = pt2.x;
    ptl.y = pt2.y;
    ptl.z = pt2.z;
    return (ptl);
}

/*-------------------------------------------------------------------------------
                                  CONNECT LINKS
---------------------------------------o0o-----------------------------------------
This function connects the links of the polygon_list so that each polygon is can
be accessed using a doublely linked list
*/

polygon_list   *connect_links(polygon)
    polygon_list   *polygon;
{
    obstacle_plane *current = polygon->polygon_start;
    int             count;

    for (; current->ccw != polygon->polygon_start; current = current->ccw)
       current->ccw->cw = current;

    polygon->polygon_start->cw = current;
    return (polygon);
}

/*-------------------------------------------------------------------------------
                                COUNT POLYGON NODES
---------------------------------------o0o-----------------------------------------
This function counts the number of nodes that make up the polygon and places the
value in the the polygon's start node structure
*/

int             count_polygon_nodes(polygon)
    polygon_list   *polygon;
{
    obstacle_plane *current = polygon->polygon_start->ccw;
    int             count = 1;

    for (; current != polygon->polygon_start;
         current = current->ccw, count++)
       if (current->ccw == NULL)
        current->ccw = polygon->polygon_start;
     return (count);
}
```

```
/*-------------------------------------------------------------------------
                              REMOVE DUPLICATE NODES
-----------------------------------------oOo-------------------------------
This function removes any duplicate nodes which are placed in the polygon link
listed when created
*/

polygon_list   *remove_duplicate_nodes(polygon)
   polygon_list   *polygon;
{
   obstacle_plane *start = polygon->polygon_start,
                  *current = start->ccw, *previous = start;
   int             count1, number_duplicate_nodes = 0;

   for (count1 = 1;
        count1 <= start->number_nodes;
        count1++, previous = current, current = current->ccw) {

      if (current->pt.x == previous->pt.x &&
          current->pt.y == previous->pt.y &&
          current->pt.z == previous->pt.z) {
        number_duplicate_nodes++;

        if (start->pt.x == current->pt.x &&
            start->pt.y == current->pt.y &&
            start->pt.z == current->pt.z) {
          polygon->polygon_start = previous;
          previous->number_nodes = current->number_nodes;
        }
        previous->ccw = current->ccw;
        free((char *) current);
        current = previous->ccw;
      }
   }
   polygon->polygon_start->number_nodes -= number_duplicate_nodes;
   return (polygon);
}

/*-------------------------------------------------------------------------
                                  REMOVE PLANES
-----------------------------------------o0o-------------------------------
This function deallocates the memory associated with the polygons formed by the
intersection of the perpendicular plane and the polyhedron
*/

polygon_list *remove_planes(start_2d)
     polygon_list *start_2d;
{
  polygon_list *previous = start_2d, *current = start_2d->next_polygon;
  obstacle_plane *node1, *node2;

  for(; current->next_polygon != NULL;
      previous = current, current = current->next_polygon){

    for(node1 = previous->polygon_start, node2 = node1->ccw;
        node2 != previous->polygon_start;
        node1 = node2, node2 = node2->ccw)
      free(node1);
    free(node1);

    free(previous);
  }
  return(NULL);
}
```

APPENDIX C

```
#include "3d_tan.h"


/*-------------------------------------------------------------------------
                    CREATE OBSTACLE LIST
-------------------------------------o0o-----------------------------------
This function is the main function used to create the 3D obstacle environ-ment. The
list of obstacles use pointers with the main list made up of major obstacles only.
Off each record in this list a pointer points to the first face of the polygon,
which in turn points at the next one.  This continues until a list of faces is
established. With in each face record is a pointer which points at a linked list
of nodes.  It calls create_face passing to it the pointer within the polygon record
that points to the first face of the polygon.  It also passes the file pointer to
the data file*/

obstacle_list  *create_obstacle_list(void)
{
   FILE           *fpt;
   int             count;
   obstacle_list  *polyhedron_list, *polygon = NULL;

   /* opens file if one, else exits program on an error */
   if ((fpt = fopen(filename, READONLY)) == NULL) {
      perror("  Error:  Data file did not open correctly.\n ");  exit(1);
   }
   else {
      read_comment(fpt);
      fscanf(fpt, "%d", &number_polygons);
      for (count = 1; count <= number_polygons; count++) {
      if (count == 1) {
         NEW_NODE(polyhedron_list);
         polygon = polyhedron_list;
      }
      else {
         NEW_NODE(polygon->next_polygon);
         polygon = polygon->next_polygon;
      }
      polygon->polyhedron_number = count;
      polygon->obstacle = create_face(polygon->obstacle, fpt);
      polygon->next_polygon = NULL;
      }
   }
   return (polyhedron_list);
}

/*-------------------------------------------------------------------------
                          CREATE FACE
-------------------------------------o0o-----------------------------------
This function creates the list off the major polygon list and makes up a list of
polygon faces.  It call create_plane, passing it the file pointer to the data file
and the pointer within each face that will point to the first node of the linked
node list.  It returns a the start record (first face in the polygon list) to the
calling function*/

polyhedron_obstacle *create_face(start_face, fpt)
   polyhedron_obstacle *start_face;
   FILE           *fpt;
{
   int             number_faces, count;
   polyhedron_obstacle *face;
```

```
   fscanf(fpt, "%d", &number_faces);
   for (count = 1; count <= number_faces; count++) {
      if (count == 1) {
      NEW_POLYHEDRON(start_face);
      face = start_face;
      }
      else {
      NEW_POLYHEDRON(face->next_face);
      face = face->next_face;
      }
      face->next_face = NULL;
      face->face_nodes = create_plane(fpt, face->face_nodes);
   }
   return (start_face);
}

/*-------------------------------------------------------------------------
                                  CREATE PLANE
-----------------------------------o0o-------------------------------------
This function creates the floatly linked list made up of the nodes of the face and
returns the start of the list to the calling function.*/

polyhedron_obstacle_plane *create_plane(fpt, start_plane)
   polyhedron_obstacle_plane *start_plane;
   FILE           *fpt;
{
   polyhedron_obstacle_plane *current = start_plane, *previous = NULL;
   int             count = 1, nodes;
   double          x, y, z;

   previous = NULL;
   fscanf(fpt, "%d", &nodes);
   for (count = 1; count <= nodes; count++) {
      fscanf(fpt, "%lf %lf %lf", &x, &y, &z);
      if (count == 1) {
        NEW_POLY_PLANE(start_plane);
        previous = start_plane;
        current = start_plane;
      }
      else {
        NEW_POLY_PLANE(current->ccw);
        current = current->ccw;
      }
      current->pt.x = x;
      current->pt.y = y;
      current->pt.z = z;
      current->cw = previous;
      current->ccw = NULL;
      previous = current;
   }
   start_plane->cw = current;
   current->ccw = start_plane;
   start_plane->number_nodes = nodes;
   return (start_plane);
}

/*-------------------------------------------------------------------------
                                  READ COMMENT
-----------------------------------o0o-------------------------------------
This function is used only to strip the comments off the polygon data file so that
they do not corrupt the link list, but an explanation of the file makeup remains
with the data file
*/

void            read_comment(fpt)
   FILE           *fpt;
{
```

```
   char            word[20];

   do
      fscanf(fpt, "%s", word);
   while (word[2] != '#');
}

/*-------------------------------------------------------------------------
                                    DISTANCE
-----------------------------------o0o-------------------------------------
This function calculates the distance from one point to another and re-turns it to
the calling function*/

double          distance(pt1, pt2)
   point           pt1, pt2;
{
   double          a, b, c, z;

   a = pt1.x - pt2.x;
   a = sqr(a);
   b = pt1.y - pt2.y;
   b = sqr(b);
   c = pt1.z - pt2.z;
   c = sqr(c);
   z = sqrt(a + b + c);

   return (z);
}

/*-------------------------------------------------------------------------
                                FIND PLANE EQUATION
-----------------------------------o0o-------------------------------------
This function finds a plane give three points.  It finds two vectors formed by
pt1pt2 and pt1pt3, then calls cross product to find the a, b and c coefficents, and
finally solves for d using a,b,c and pt1 */

equation        find_plane_equation(pt1, pt2, pt3)
   point           pt1, pt2, pt3;
{
   point           vector1_2, vector1_3;
   equation        pl_eq;

   vector1_2.x = pt2.x - pt1.x;
   vector1_2.y = pt2.y - pt1.y;
   vector1_2.z = pt2.z - pt1.z;
   vector1_3.x = pt3.x - pt1.x;
   vector1_3.y = pt3.y - pt1.y;
   vector1_3.z = pt3.z - pt1.z;

   pl_eq = cross_product(vector1_2, vector1_3);
   pl_eq.d = ((pl_eq.x * (-1.0) * pt1.x) + (pl_eq.y * (-1.0) * pt1.y));
   pl_eq.d = pl_eq.d + (pl_eq.z * (-1.0) * pt1.z);
   return (pl_eq);
}

/*-------------------------------------------------------------------------
                                  CROSS PRODUCT
-----------------------------------o0o-------------------------------------
This function finds the cross product of two vectors and returns the coefficients
to the calling function.*/

equation        cross_product(vector1, vector2)
   point           vector1, vector2;
{
   equation        plane;

   plane.x = (vector1.y * vector2.z) - (vector1.z * vector2.y);
   plane.y = -1.0 * ((vector1.x * vector2.z) - (vector1.z * vector2.x));
```

```
   plane.z = (vector1.x * vector2.y) - (vector1.y * vector2.x);

   return (plane);
}

/*-------------------------------------------------------------------------
                              INTERSECT POLYHEDRON
---------------------------------o0o-----------------------------------------
This function determines if the line formed from two points intersect a polyhedron.
If so, it returns 1, else it returns zero.  It calls find_plane_equation, passing
it three nodes on the current face being checked and receives the plane equation.
It then calls inter-section_point to determine if the line intersects the plane.
This function returns the point if there is an intersection, else it sets the x
coeffi-cient = -999999.  If there is an intersection point, the function call
lines_intersect, which determines if the intersection is within the boun-dries of
the plane.
*/

int             intersect_polyhedron(pt1, pt2)
   point           pt1, pt2;

{
   obstacle_list  *current_obstacle;
   polyhedron_obstacle *face;
   polyhedron_obstacle_plane *node;
   equation        pl_eq;

   for (current_obstacle = polyhedron_list; current_obstacle != NULL;
        current_obstacle = current_obstacle->next_polygon) {

      for (face = current_obstacle->obstacle; face != NULL;
           face = face->next_face) {

        node = face->face_nodes;
        pl_eq = find_plane_equation(node->pt, node->ccw->pt,
                                    node->ccw->ccw->pt);

        if ((pl_eq.x * pt1.x + pl_eq.y * pt1.y + pl_eq.z * pt1.z + pl_eq.d) == 0
             ||
            (pl_eq.x * pt2.x + pl_eq.y * pt2.y + pl_eq.z * pt2.z + pl_eq.d) == 0)
          continue;       /* if pt1 lies on the plane, go to next plane */

        intersection = intersection_point(pt1, pt2, pl_eq);

        if (intersection.x != NO_INTERSECTION) {
          if (!lines_intersect(intersection, face->face_nodes, pl_eq))
            return (1);
        }
      }
   }
   return (0);
}

/*-------------------------------------------------------------------------
                              INTERSECTION POINT
---------------------------------o0o-----------------------------------------
This function finds the intersection of a plane and a line.  It accepts three
arguments, 2 points (the line) and an equation for the plane.  It returns the
intersection point if there is one or it sets the plane.x value equal to -99999.0.
This value is a flag that lets the calling function know that there is no
intersection point and the line and plane are parallel.*/

point           intersection_point(pt1, pt2, plane)
   point           pt1, pt2;
   equation        plane;
{
   double          y = 0.0, t = 0.0;
```

```
   y = (plane.x * (pt2.x - pt1.x) + plane.y * (pt2.y - pt1.y));
   y = y + (plane.z * (pt2.z - pt1.z));
   if (y == FALSE) {
      intersection.x = NO_INTERSECTION;
      return (intersection);
   }
   t = (plane.x * pt1.x + plane.y * pt1.y + plane.z * pt1.z + plane.d);
   t = -1 * t / y;

   if (0.0 <= t && t <= 1.0) {
      intersection.x = pt1.x + (pt2.x - pt1.x) * t;
      intersection.y = pt1.y + (pt2.y - pt1.y) * t;
      intersection.z = pt1.z + (pt2.z - pt1.z) * t;
   }
   else
      intersection.x = NO_INTERSECTION;

   return (intersection);
}

/*-------------------------------------------------------------------------
                    LINES INTERSECT for INSIDE POLYGON CHECK
-----------------------------------o0o-------------------------------------
This function determines if a point is within the boundries of a polygon. If the
point is, the function returns 1, else it returns 0.  Since the plane can be on any
plane in the 3d environment, the function must check each plane face (xy, xz, yz).
*/

int             lines_intersect(intersect, start, pl_eq)
   point           intersect;
   polyhedron_obstacle_plane *start;
   equation        pl_eq;
{
   polyhedron_obstacle_plane *current = start;
   int             number_nodes = start->number_nodes, count;
   int             intersect_count = 0;
   double          t, s, b, c;
   point           temp_pt;

   temp_pt = find_point_on_obstacle_plane_face(intersect, pl_eq);

   for (count = 1;  count <= number_nodes; count++, current = current->ccw) {

      /* CHECK X Y PLANE */

      b = (temp_pt.x - intersect.x) * (current->ccw->pt.y - current->pt.y);
      c = (temp_pt.y - intersect.y) * (current->ccw->pt.x - current->pt.x);
      b = b - c;

      if (b != 0) {
        s = (intersect.y - current->pt.y) * (current->ccw->pt.x - current->pt.x);
        s -= (intersect.x - current->pt.x) * (current->ccw->pt.y - current->pt.y);
        s = s / b;
        t = (intersect.y - current->pt.y) * (temp_pt.x - intersect.x);
        t = t - (intersect.x - current->pt.x) * (temp_pt.y - intersect.y);
        t = t / b;

        if (0 <= s && s <= 1 && 0 <= t && t <= 1) {
          intersect_count++;
          continue;
        }
      }

      /* CHECK X Z PLANE */
      b = (temp_pt.y - intersect.y) * (current->ccw->pt.z - current->pt.z);
      c = (temp_pt.z - intersect.z) * (current->ccw->pt.y - current->pt.y);
      b = b - c;
```

```
      if (b != 0) {
        s = (intersect.z - current->pt.z) * (current->ccw->pt.y - current->pt.y);
        s -= (intersect.y - current->pt.y) * (current->ccw->pt.z - current->pt.z);
        s = s / b;
        t = (intersect.z - current->pt.z) * (temp_pt.y - intersect.y);
        t = t - (intersect.y - current->pt.y) * (temp_pt.z - intersect.z);
        t = t / b;

        if (0 <= s && s <= 1 && 0 <= t && t <= 1) {
          intersect_count++;
          continue;
        }
      }

      /* CHECK Y Z PLANE */
      b = (temp_pt.x - intersect.x) * (current->ccw->pt.z - current->pt.z);
      c = (temp_pt.z - intersect.z) * (current->ccw->pt.x - current->pt.x);

      if (b - c >= 1e-3) {
        s = (intersect.z - current->pt.z) * (current->ccw->pt.x - current->pt.x);
        s -= (intersect.x - current->pt.x) * (current->ccw->pt.z - current->pt.z);
        s = s / (b - c);
        t = (intersect.z - current->pt.z) * (temp_pt.x - intersect.x);
        t = t - (intersect.x - current->pt.x) * (temp_pt.z - intersect.z);
        t = t / (b - c);

        if (0 <= s && s <= 1 && 0 <= t && t <= 1) {
          intersect_count++;
          continue;
        }
      }
   }
   if (intersect_count % 2 == 0)
      return (1);
   else
      return (0);
}

/*-------------------------------------------------------------------------
                        FIND POINT ON OBSTACLE PLANE FACE
-------------------------------------o0o-----------------------------------
This function finds a point that is on the plane.  It is called by the function
that checks if the intersection of the ray formed by two points intersects any
polyhedrons in the environment.*/

point           find_point_on_obstacle_plane_face(intersect, plane)
   point           intersect;
   equation        plane;
{
   point           pt;

   pt.z = -5;
   if ((plane.x == 0 || plane.y == 0) && plane.z == 0) {
      pt.x = intersect.x;
      pt.y = intersect.y;
   }
   else {
      pt.y = 17;
      if (plane.x != 0)
        pt.x = -(plane.d + plane.z * pt.z + plane.y * pt.y) / plane.x;
      else
        pt.x = 50;
   }
   return (pt);
}
```

```
/*-------------------------------------------------------------------------
                        VERTICAL AND PERPENDICULAR PLANES
-----------------------------------o0o-------------------------------------
This function finds the vertical and perpendicular planes formed by two points.
*/

void            vertical_n_perpendicular_plane(pt1, pt2)
   point           pt1, pt2;
{
   point           pt3;

   pt3.x = pt2.x;
   pt3.y = pt2.y;
   pt3.z = pt2.z + 1;
   vertical = find_plane_equation(pt1, pt2, pt3);
   if (pt1.x != pt2.x) {
      pt3.x = pt2.x;
      pt3.y = pt2.y + 1;
   }
   else {
      pt3.x = pt2.x + 1;
      pt3.y = pt2.y;
   }
   pt3.z = pt2.z;
   perpendicular = find_plane_equation(pt1, pt2, pt3);
}
```

APPENDIX D

```
#include "3d_tan.h"

/*-----------------------------------------------------------------
                          CHECK ACTIVE NODE LIST
-----------------------------------OOO-------------------------------
This function checks the active node list to determine if the expanded
node found in find_tangents_from_inner_nodes has already been used as a
node to an existing path.  If the point has been used, the function deter-
mines if the possible new path has a cost less than the existing path.  If
the cost is less, the function calls the function mark_old_path_with_node_
unexpandable, which marks the path as unexpandable.  If the new possible
path has a smaller cost, then the function returns a one (1) else it
returns a zero (0).
*/

int             check_active_node_list(head_active_nodes, shortest, pt)
   active_nodes   *head_active_nodes;
   paths          *shortest;
   point           pt;
{
   active_nodes   *current = head_active_nodes;
   a_path         *last_node = shortest->last_node;

   for (; current != NULL; current = current->next_active_node) {
      if (current->pt.x == pt.x && current->pt.y == pt.y && current->pt.z == pt.z)
{
        if (current->distance > (shortest->cost +
                 distance(shortest->last_node->pt, pt))) {
           start_path =
                mark_old_path_with_node_unexpandable(current->when_path_found);
           return (1);
        }
        else
           return(0);
      }
   }
   return(1);
 }

/*-----------------------------------------------------------------
                       MARK OLD PATH WITH NODE UNEXPANDABLE
-----------------------------------OOO-------------------------------
This functions is called if the existing path has been determined as too
expensive (ie; there is a node in the path that can be reached by another
path with a smaller cost).  The function searches the path list to find
the correct path and then marks the flag "able2expand" as FALSE (0).
*/

paths          *mark_old_path_with_node_unexpandable(path_number)
   int              path_number;
{
   paths           *current = start_path;

   for (; current != NULL; current = current->next_path) {
      if (current->when_path_found == path_number)
      current->able2expand = FALSE;
   }
   return (start_path);
```

```
}

/*-------------------------------------------------------------------
                    INSERT NODES INTO ACTIVE NODE LIST
-----------------------------------OOO-----------------------------*/
active_nodes   *insert_nodes_into_active_node_list(head_node_list, pt,
                                                   cost)
   active_nodes   *head_node_list;
   point           pt;
   double          cost;
{
   static active_nodes *last_node_added = NULL;

   if (head_node_list == NULL) {
      NEW_ACTIVE_NODE(head_node_list);
      last_node_added = head_node_list;
      last_node_added->next_active_node = NULL;
   }
   else {
      NEW_ACTIVE_NODE(last_node_added->next_active_node);
      last_node_added = last_node_added->next_active_node;
      last_node_added->next_active_node = NULL;
   }
   last_node_added->pt = assign_point_values(pt);
   last_node_added->distance = cost;
   last_node_added->when_path_found = number_paths_found;
   return (head_node_list);
}
```

APPENDIX E

This appendix contains the source code I began to implement for going up, around and up type of paths as described in Chapter IV. The file over&@.c contains the source code developed todate and not yet implemented

```
#include "3d_tan.h"
#include "plot.h"

#define OVER 0

paths *up_around_up(path, tan, start, goal)
   paths *path;
   int tan;
{/*
  polyhedron_face *from_face = NULL, *to_face = NULL;
  obstacle_list *polyhedron1, *polyhedron2;
  a_path *tangent_list;
  vertical_path *vertical_list;
  int count;
  polygon_list *polygons, *recursive_polygons = NULL;
  paths *partial_path = NULL, *shortest, *short_list;

  for(polyhedron1 = polyhedron_list;
      polyhedron1 != NULL;
      polyhedron1 = polyhedron1->next_polygon){
      from_face = find_polyhedron_face(polyhedron1);

    for(polyhedron2 = polyhedron_list;
     polyhedron2 != NULL;
     polyhedron2 = polyhedron2->next_polygon){

      if(polyhedron2->polyhedron_number == polyhedron1->polyhedron_number ||
       polyhedron2->polyhedron_number == path->polygon4vert)
     continue;
      else
     to_face = find_polyhedron_face(polyhedron2);

      /*since tangent returns a tangent between two polygons, have to form a
      polygon list with only the to_face and from face in the list so that a
      the tangent is found*/
      polygons = make_list(from_face, to_face);
      for(count = 1; count <= 4; count++){
     if(NULL == (tangent_list = tangent(path, from_face, to_face->face_nodes,
                  (count ==1||3?MINUS:PLUS), polygons, OVER,
                  count ==1||2?MINUS:PLUS))) printf("you blew it caddell");

     if(!chech_tangent_list(tangent_list, path)) continue;

     if(!intersect_polyhedron(tangent_list->pt, tangent_list->next_node->pt))
       continue;
     else{
       vertical_list    =    find_vertical_nodes(tangent_list->next_node->pt,
tangent_list->pt);
       vertical_n_perpendicular_plane(vertical_list->next_node->next_node->pt,
path->last_node->pt);
       recursive_polygons    =    build_two_d_polygon_list(recursive_polygons,
perpendicular, FALSE);
       partial_paths = duplicate_path(path);
       partial_paths->plane_number = plane_number;
```

```
        partial_paths = continue_vertical_path(partial_paths, recursive_polygons


        recursive_polygons = free_polygon_list(recursive_polygons);
      }
       }/*end count for structure*/
     } /*end inner for structure*/
   }  /*end outer for structure*/

   free_pseudo_polygon_list(polygons);
   free_tangent_list(tangent_list);
*/
   return(path);
}


polyhedron_face *find_polyhedron_face(polyhedron)
   obstacle_list *polyhedron;
{
   polyhedron_face *face, *min_vert_face = NULL;
   obstacle_plane *node, *min_vert_node = NULL;

   for(face = polyhedron->obstacle;
       face != NULL;
       face = face->next_face){

     node = face->face_nodes;
     if(node->pt.z != node->ccw->pt.z &&
        node->pt.z != node->cw->pt.z) continue;
     else{
       if(NULL == min_vert_face) min_vert_face = face;
       else{
      if(min_vert_face->face_nodes->pt.z < node->pt.z) break;
      else min_vert_face = face;
       }
     }
   }

   return(min_vert_face);
}


polygon_list *make_list(to, from)
   obstacle_plane *to, *from;
{
   polygon_list *polygons;

   NEW_POLYGON(polygons);
   NEW_POLYGON(polygons->next_polygon);

   polygons->polygon_start = from;
   polygons->next_polygon->polygon_start = to;
   polygons->next_polygon->next_polygon = NULL;

   return(polygons);
}

int free_pseudo_polygon_list(polygons)

   polygon_list *polygons;

{

   free(polygons->next_polygon);
   free(polygons);
   return(1);
}
```

```
int chech_tangent_list(tangent_list, path)
  a_path *tangent_list;
  paths  *path;

{
  double dir;

  if(path->forward) dir = goal_direction;
  else dir = norm(goal_direction - HPI);

  if(ABS(norm(dir -
         atan2(tangent_list->pt.y - path->last_node->pt.y,
         tangent_list->pt.x - path->last_node->pt.x))) >=HPI)
    return(0);

  if(abs(norm(dir -
      atan2(tangent_list->next_node->pt.y - path->last_node->pt.y,
      tangent_list->next_node->pt.x - path->last_node->pt.x))) >= HPI)
    return(0);

  return(1);
}

a_path *free_tangent_list(list)
  a_path *list;
{
  a_path *previous, *current;

  for(current = list, previous = list;
      current != NULL;
      previous = current, current = current->next_node, free(previous))
    free(previous);
  return(NULL);
}

polygon_list *free_polygon_list(list)
  polygon_list *list;
{
  polygon_list *previous_poly, *current_poly;
  obstacle_plane *node;
  int node_number, count;

  for(current_poly = list, previous_poly = list;
      NULL != current_poly;
      previous_poly = current_poly, current_poly = current_poly->next_polygon,
     free(previous_poly)){

    for(node = current_poly->polygon_start,
        node_number = current_poly->polygon_start->number_nodes,
        count = 1;
        count <= node_number;
        node = node->ccw, node->cw->ccw = NULL, free(node->cw), count++);
  }
  free(previous_poly);
  return(NULL);
}
```

APPENDIX F

This appendix contains the source code found in path.c

```
#include "3d_tan.h"
#include "plot.h"

#define LOOPFOREVER 1
#define AROUND_PATH 1

/*-------------------------------------------------------------------------
                                REMOVE PATHS
-----------------------------------o0o-------------------------------------
This function frees from memory the created possible paths so that another
iteration of path building can commence.  It accepts the head of the list
and returns a NULL pointer to the head of the list.*/

paths           *remove_paths(start, shortest)
   paths           *start, *shortest;
{
   paths           *current = start->next_path, *previous = start;
   a_path          *node, *node1;
   int              i;

   for (previous = start, current = start->next_path;
     current->next_path != NULL;
     previous = current, current = current->next_path) {

      if (shortest_path == previous) {
      shortest->next_path = NULL;
      continue;
      }

      for (node = previous->last_node, node1 = node->next_node;
        node1 != NULL;
        node = node1, node1 = node1->next_node)
      free(node);

      free(node);          /* frees last node in the path being deleted */
      free(previous);
   }
   if (shortest_path != current) {
      previous = current;

      for (node = previous->last_node, node1 = node->next_node;
        node1 != NULL;
        node = node1, node1 = node1->next_node)
      free(node);

      free(node);          /* frees last node in the path being deleted */
      free(previous);      /* frees last path in the path list */
   }
   start = shortest;
   return (start);
}
```

```
/*-------------------------------------------------------------------------
                                    ORDER
-----------------------------------o0o-----------------------------------
This function uses the area function for a triangle in determining if the
current node is the common tangent.  It uses one of the two adjacent nodes
and the inputted point to form the triangle.  It returns the value found
(double) in the calculations.*/

double          order(p1, current, adjacent_node)
   point           p1;
   obstacle_plane *current, *adjacent_node;
{
   double          z;

   z = (current->pt.x - p1.x) * (adjacent_node->pt.y - p1.y);
   z = z - (adjacent_node->pt.x - p1.x) * (current->pt.y - p1.y);
   return (z);
}

/*-------------------------------------------------------------------------
                              START FINDING PATHS
-----------------------------------o0o-----------------------------------
This function finds all the valid tangential lines from the start point
to polygons in the start_2d list and places them in the start_paths link
list.  It returns this link list to the calling function.  The function
calls plus and minus tangent functions to find the tangents from the start
to all the polygons in the start_2d list.  As it finds the two tangents
for a polygon, it calls cross_polygon to determine if there is not a poly-
gon between the point and the start point.  If not, then the node is added
to the list as the beginning of a valid path, by calling create_path_n_nodes.
It calls assign_point_values, which places the new point into the link list.
The function also calls the function atan2, which is a library function that
determines the direction from the start point to the node added to the
start_paths list.*/

paths          *start_finding_paths(start_paths, start_twod, start_pt, goal,
                                    boundry, active_node_list)
   paths           *start_paths;
   polygon_list    *start_twod;
   point            start_pt, goal;
   boundries        boundry;
   active_nodes    *active_node_list;

{
   paths           *current_path = NULL;
   polygon_list    *current_polygon;
   obstacle_plane  *current_node;
   int              count, tangent;
   point            pt1, pt2;

   plane_number = 1;       /* tells what plane the 2d representation is
                           * on */

   vertical_n_perpendicular_plane(start_point, goal);
   /*start_2d is a global defining the two d polygon list*/
   start_2d = build_two_d_polygon_list(start_2d, perpendicular, FALSE);

#ifdef DOS
   if (plot2screen)
      data_plot(start_2d);
#endif

   for (current_polygon = start_2d;
     current_polygon != NULL;
     current_polygon = current_polygon->next_polygon) {

      pt1 = plus_tangent(current_polygon->polygon_start, start_pt);
      pt2 = minus_tangent(current_polygon->polygon_start, start_pt);
```

```
      if (direction_heuristic_from_start(start_pt, pt1) &&
       boundry_check(boundry, pt1)) {
      if(!cross_polygon(start_2d, start_pt, pt1)){
        start_paths = create_path_n_nodes(start_paths, current_path);
        tangent = PLUS;
        if (current_path == NULL)
           current_path = start_paths;
        else
           current_path = current_path->next_path;
         current_path = update_current_path(current_path, /*start_paths,*/
                              tangent, start_pt, pt1, goal,
                              current_polygon->polygon_number);
         current_path = add_start_point_to_path(current_path, start_pt);

#ifdef DOS
         if (plot2screen)
            draw_line_segment(start_pt, current_path->last_node->pt, YELLOW,
                              REGULAR);
#endif
       }
       else{
         if(!horizontal){
           start_paths = up_and_over(start_paths, current_path, start_point,
                                     pt1, start_point, PLUS,
                          current_polygon->polyhedron_number);
           if(current_path == NULL) current_path = start_paths;
           else current_path = current_path->next_path;
           current_path->forward = YES;
         }
       }
    }
    if (direction_heuristic_from_start(start_pt, pt2) &&
         boundry_check(boundry, pt2)) {
       if(!cross_polygon(start_2d, start_pt, pt2)){
         start_paths = create_path_n_nodes(start_paths, current_path);
         tangent = MINUS;
         if (current_path == NULL)
            current_path = start_paths;
         else
           current_path = current_path->next_path;
         current_path = update_current_path(current_path, tangent, start_pt,
                                            pt2, goal,
                                            current_polygon->polygon_number);
         current_path = add_start_point_to_path(current_path, start_pt);

#ifdef DOS
         if (plot2screen)
            draw_line_segment(start_pt, current_path->last_node->pt, YELLOW,
                                    REGULAR);
#endif
        }
        else{
          if(!horizontal){
            start_paths = up_and_over(start_paths, current_path, start_point,
                                      pt2, start_point, MINUS,
                                      current_polygon->polyhedron_number};
            if(current_path == NULL) current_path = start_paths;
            else current_path = current_path->next_path;
            current_path->forward = YES;
          }
        }
     }
   }

   /*this function finds the one partial vertical path from the start
     point over all obstacles to goal*/
   start_paths = up_and_over(start_paths, current_path, start_point, goal,
                   start_point, VERTICAL);
```

```
   current_path->next_path->forward = YES;  /*current_path->next is path
                                added to the list that is
                                up and over all obstacles*/
   current_path = current_path->next_path;
   /*this function finds all the partial vertical paths from the goal, thus
     working backwards to find all the paths which go around one or more
     obstacles and over at least one obstacles*/
   start_paths = up_and_over_from_goal(start_paths, current_path, goal, start_2d);

   return (start_paths);
}


/*-------------------------------------------------------------------------
                            ADD START POINT TO PATH
-----------------------------------o0o-----------------------------------
This function adds the start point to the partial list being created by
the function start finding paths.*/

paths          *add_start_point_to_path(current_path, start_point)
   paths          *current_path;
   point           start_point;
{
   a_path         *current_node;

   for (current_node = current_path->last_node;
     current_node->next_node != NULL;
     current_node = current_node->next_node);

   NEW_NODES(current_node->next_node);
   current_node->next_node->pt = assign_point_values(start_point);
   current_node->next_node->next_node = NULL;
   current_node->next_node->next_after_node = current_node;
   return (current_path);
}


/*-------------------------------------------------------------------------
                            CREATE PATH AND NODES
-----------------------------------o0o-----------------------------------
This function is called to create and insert into the start_paths link
list new points which form valid paths from the start.*/

paths          *create_path_n_nodes(start_paths, current_path)
   paths          *start_paths, *current_path;
{
   if (start_paths == NULL) {
      NEW_PATH(start_paths);
      current_path = start_paths;
   }
   else {
      NEW_PATH(current_path->next_path);
      current_path = current_path->next_path;
   }
   current_path->polygon4vert_PTR = NULL;
   current_path->next_path = NULL;
   NEW_NODES(current_path->last_node);
   current_path->last_node->next_node = NULL;
   return (start_paths);

}
```

```
/*-------------------------------------------------------------------------
                                CROSS POLYGON
-------------------------------------o0o-----------------------------------
This function determines if a line crosses the polygon.  It checks each
line segment of the polygon and returns 1 if there is an intersecttion.
The second for structure is used to check if p2 is on a polygon.  If so,
boolean is asserted to true (int value = 1).  This tells the rest of the
for loop to go on to the next polygon*/

int             cross_polygon(start, p1, p2)
   polygon_list   *start;
   point           p1, p2;
{
   int             count, boolean;
   polygon_list   *polygon = NULL;
   obstacle_plane *current = NULL;
   point           p3, p4;
   double          w, x, y, z;

   for (polygon = start; polygon != NULL;
         polygon = polygon->next_polygon) {

      for (count = 1, current = polygon->polygon_start, boolean = FALSE;
        count <= polygon->polygon_start->number_nodes;
        count++, current = current->ccw) {
       if (p2.x == current->pt.x && p2.y == current->pt.y) {
          boolean = TRUE;
          break;
       }
   }

   if (!boolean) {
      for (count = 1, current = polygon->polygon_start;
           count <= polygon->polygon_start->number_nodes;
           count++, current = current->ccw) {
         p3.x = current->pt.x, p3.y = current->pt.y;
         p4.x = current->ccw->pt.x, p4.y = current->ccw->pt.y;
         w = sign(function_one(p3.x, p3.y, p1, p2));
         x = sign(function_one(p4.x, p4.y, p1, p2));
         y = sign(function_two(p1.x, p1.y, p3, p4));
         z = sign(function_two(p2.x, p2.y, p3, p4));

         if (w != x && y != z) {

            return (1);   /* line intersects polygon */
         }
       }
       current = current->ccw;
    }
  }
  return (0);
}

/*-------------------------------------------------------------------------
                        FUNCTION ONE AND FUNCTION TWO
-------------------------------------o0o-----------------------------------
Funcitons one and two are simple line equaitons used to determine if two
lines intersect.  They both return the value of the equation to the
calling funcion*/

double          function_one(x, y, p1, p2)
   double          x, y;
   point           p1, p2;
{
   int             z = 0;

   z = ((y - p1.y) * (p2.x - p1.x)) - ((p2.y - p1.y) * (x - p1.x));
   return (z);
```

```
}

double          function_two(x, y, p3, p4)
   double          x, y;
   point           p3, p4;
{
   int             z = 0;

   z = ((y - p3.y) * (p4.x - p3.x)) - ((p4.y - p3.y) * (x - p3.x));
   return (z);
}

/*-------------------------------------------------------------------------
                                DUPLICATE PATH
-------------------------------------o0o-----------------------------------
This function accepts from the calling function a valid path, duplicates
it and inserts the duplication into the start_paths link list.  It calls
duplicate_entries to duplicate the header data of each path (ie type of
tangent, total cost, estimated cost, direction etc...)*/

paths           *duplicate_paths(current)
   paths           *current;
{
   a_path          *node, *duplicate_node;
   paths           *duplicate;

   duplicate = duplicate_entries(current);
   for (node = current->last_node;
     node != NULL;
     node = node->next_node) {
      if (duplicate->last_node == NULL) {
      NEW_NODES(duplicate->last_node);
      duplicate_node = duplicate->last_node;
      duplicate_node->next_after_node = NULL;
      }
      else {
      NEW_NODES(duplicate_node->next_node);
      duplicate_node->next_node->next_after_node = duplicate_node;
      duplicate_node = duplicate_node->next_node;
      }
      duplicate_node->next_node = NULL;
      duplicate_node->polygon_number = node->polygon_number;
      duplicate_node->pt = assign_point_values(node->pt);
   }
   return (duplicate);
}

/*-------------------------------------------------------------------------
                                DUPLICATE ENTRIES
-------------------------------------o0o-----------------------------------
This function duplicates the major entries of a valid path needed in the
calculations for finding the best path*/

paths           *duplicate_entries(current)
   paths           *current;
{
   paths           *duplicate;

   NEW_PATH(duplicate);
   duplicate->last_node = NULL;
   duplicate->cost = current->cost;
   duplicate->estimated_cost = current->estimated_cost;
   duplicate->total_cost = current->total_cost;
   duplicate->ray_direction = current->ray_direction;
   duplicate->plane_number = current->plane_number;
   duplicate->type_tangent = current->type_tangent;
   duplicate->forward = current->forward;
   duplicate->polygon4vert_PTR = NULL;
```

```
   return (duplicate);
}

/*-------------------------------------------------------------------------
                              EXTEND PATH
-----------------------------------o0o-------------------------------------
This function accepts three parameters (the start path for either possible
verticle paths of possible perpendicular paths, the start pointer for the
2d polygon representation, and the goal point.  It searches the possible
path list for the shortest total distance from start to goal.  It checks
the line projection of this path and the goal with the goal point for any
intersection with polygons from the polygon list (start_2d list).  If no
intersections, the goal is appended to this shortest path and the short-
est path is returned to the the calling funcition.  If there is an inter-
section, the function calls FIND_TANGENTS FROM_INNER_NODES, passing to it
the shortest path and the start_2d pointer
*/

paths          *extend_path(start, goal, shortest_path, start_2d, boundry,
                                active_node_list)
   polygon_list   *start_2d;
   point           start, goal;
   paths          *shortest_path;
   boundries       boundry;
   active_nodes   *active_node_list;
{
   paths          *current_path = start_path;
   a_path         *new_record, *current;

   if (shortest_path->type_tangent != NONE) {
      shortest_path = find_tangents_from_inner_node(shortest_path, start_2d,
                           boundry, active_node_list,
                             1);
      shortest_path->able2expand = NO;

   }
   else {
      if(shortest_path->forward == YES) /*partial path is from start or
                           forward*/
          start_2d = add_on_new_plane_intersection(start_2d,
                           shortest_path->last_node,
                           goal);
      else  /*partial path is from the goal or backwards*/
         start_2d = add_on_new_plane_intersection(start_2d,
                           shortest_path->last_node,
                           start);

      shortest_path->plane_number = plane_number;
      shortest_path = up_around_up(shortest_path, PLUS, start_point, goal);

      shortest_path = continue_vertical_path(shortest_path, start_2d);
   }
   return (shortest_path);
}

/*-------------------------------------------------------------------------
                              GOAL REACHED UPDATE
-----------------------------------o0o-------------------------------------
This function appends the goal point to the shortest path if the last node
of the shortest path list and the goal point are visible to each other.
it then returns the entire list of possible paths back to the calling
function.*/

paths          *goal_reached_update(goal, shortest, partial_path)
   point           goal;
   paths          *shortest;
   a_path         *partial_path;
{
```

```
   a_path          *new_record, *temp;
   double           cost = 0;

#ifdef DOS
   if (!plot2screen)
      printf("\nYou have reached the goal\n");
#elif
      printf("\nYou have reached the goal\n");

#endif

   for(temp = partial_path; temp->next_node != NULL;
         temp = temp->next_node)
          cost += distance(temp->pt, temp->next_node->pt);

   temp->next_node = shortest->last_node;
   shortest->last_node = partial_path;
   cost += distance(temp->pt, temp->next_node->pt);
   shortest->cost += cost;

   NEW_NODES(new_record);
   new_record->pt = assign_point_values(goal);
   shortest->cost += distance(shortest->last_node->pt, new_record->pt);
   new_record->next_node = shortest->last_node;
   shortest->last_node = new_record;
   shortest->total_cost = shortest->cost;
   shortest->estimated_cost = 0;
   shortest->type_tangent = NONE;

   return (shortest);
}


/*-------------------------------------------------------------------------
                        FIND_TANGENTS_FROM_INNER_NODES
-----------------------------------o0o-----------------------------------
This function finds all valid tangents from an inner node to other poly-
gons.  It is very similiar to the start_finding_paths function; however
it checks the direction of the possible new node of the path with the
current direction of the path to determine if the node is valid.  It also
calls a variation of cross_polygon (cross_polygon_node_on) that takes
into consideration that both the node that form a line lie on a polygon.*/

paths          *find_tangents_from_inner_node(shortest, start_2d, boundry,
                                   head_node_list, recursion_number)
   paths           *shortest;
   polygon_list    *start_2d;
   boundries        boundry;
   active_nodes    *head_node_list;
   int              recursion_number;
{
   polygon_list    *polygon, *previous, *polygon1;
   paths           *expanded, *same_polygon = NULL;
   a_path          *new_record;

#ifdef DOS
   a_path          *print_path;
#endif

   point            pt1, pt2;
   a_path          *current = shortest->last_node,
                   *minus_tangent_list,
                   *plus_tangent_list;

   for (polygon1 = start_2d;/* finds the polygon the last node of the
                             * shortest path lies on */
     polygon1->polygon_number != shortest->last_node->polygon_number ||
     polygon1->plane_number != shortest->plane_number;
```

```
      polygon1 = polygon1->next_polygon);

    for (polygon = start_2d;
      polygon != NULL;
      polygon = polygon->next_polygon) {

       on_same_polygon = FALSE;
       if(shortest->plane_number < polygon->plane_number)
      break;  /*planes found after the plane which shortest path last node
            is are not checked*/
       if (shortest->plane_number != polygon->plane_number ||
        shortest->last_node->polygon_number == polygon->polygon_number)
       continue;


       minus_tangent_list = tangent(shortest, polygon1, polygon->polygon_start,
                        MINUS, start_2d, AROUND_PATH);
       plus_tangent_list = tangent(shortest, polygon1, polygon->polygon_start,
                       PLUS, start_2d, AROUND_PATH);


       if (direction_heuristic4tangent(shortest, plus_tangent_list) &&
           boundry_check(boundry, plus_tangent_list->pt) &&
          boundry_check(boundry, plus_tangent_list->next_node->pt)) {
       if(NULL != plus_tangent_list &&
         check_active_node_list(head_active_nodes, shortest,
                       plus_tangent_list->pt)) {
#ifdef DOS
          if (plot2screen){
            for(print_path = plus_tangent_list; print_path->next_node != NULL;
                print_path = print_path->next_node)
               draw_line_segment(print_path->pt, print_path->next_node->pt, YELLOW,

                                       REGULAR);
            draw_line_segment(print_path->pt, shortest->last_node->pt, YELLOW,
                    REGULAR);
          }
#endif

        paths_found++;
        expanded = duplicate_paths(shortest);
        shortest = expand_paths_with_values(shortest, expanded,
                           plus_tangent_list, head_node_list,
                              polygon->polygon_number, PLUS);
        if (on_same_polygon)
        same_polygon = expanded;
       }
       }

       if (direction_heuristic4tangent(shortest, minus_tangent_list) &&
           boundry_check(boundry, minus_tangent_list->pt) &&
          boundry_check(boundry, minus_tangent_list->next_node->pt)) {
       if(NULL != minus_tangent_list &&
          check_active_node_list(head_active_nodes, shortest,
                       minus_tangent_list->pt)) {
#ifdef DOS
          if (plot2screen){
            for(print_path = minus_tangent_list; print_path->next_node != NULL;
                print_path = print_path->next_node)
              draw_line_segment(print_path->pt, print_path->next_node->pt, YELLOW,
                               REGULAR);
            draw_line_segment(print_path->pt, shortest->last_node->pt, YELLOW,
                               REGULAR);
          }
#endif
          paths_found++;
          expanded = duplicate_paths(shortest);
```

```
         shortest = expand_paths_with_values(shortest, expanded,
                           minus_tangent_list, head_node_list,
                              polygon->polygon_number, MINUS);
         if (on_same_polygon)
            same_polygon = expanded;
       }
      }
   }
   return (shortest);
}

/*-------------------------------------------------------------------------
     -                      CHECK FOR NODE ON POLYGON
-------------------------------------o0o-----------------------------------
This function determines if a point is on a polygon.  Since all the
polygons lie on the same plane, depth was not considered in this function
and the z value of the point was not used.  It returns a one if the node
is on a polygon, else a zero*/

int              check_for_node_on_polygon(pt, current)
   point            pt;
   obstacle_plane *current;
{
   if (pt.x == current->pt.x &&
       pt.y == current->pt.y &&
       pt.z == current->pt.z)
      return (1);

   if (pt.x == current->ccw->pt.x &&
       pt.y == current->ccw->pt.y &&
       pt.z == current->ccw->pt.z)
      return (1);
   else
      return (0);
}

/*-------------------------------------------------------------------------
                      REMOVE SHORTEST PATH
-------------------------------------o0o-----------------------------------
This function removes, clears from memory the path marked as the shortest.
In doing so, it reestablishes the link list and returns to the calling
function the start of the path list.*/

paths           *remove_shortest_path(shortest, start_path)
   paths           *shortest, *start_path;
{
   paths           *current, *previous;
   a_path          *current_node, *previous_node;

   for (current = start_path;
     current != shortest;
     previous = current, current = current->next_path);

   if (current == start_path)
      start_path = current->next_path;
   else {
      previous->next_path = current->next_path;
   }
   current->next_path = NULL;
   previous_node = current->last_node;
   current_node = previous_node;
   free(current);
   do {
      current_node = current_node->next_node;
      free(previous_node);
      previous_node = current_node;
   } while (previous_node != NULL);
```

```
   return (start_path);
}

/*-------------------------------------------------------------------------
                          CROSS POLYGON NODE ON
-------------------------------------o0o---------------------------------
This function determines if a line crosses the polygon.  It checks each
line segment of the polygon and returns 1 if there is an intersection.
The first for structure is used to check if p2 is on a polygon.  If so,
boolean is asserted to true (int value = 1).  This tells the rest of the
for loop to go on to the next polygon*/

int             cross_polygon_node_on(start, p1, p2, plane_number)
   polygon_list   *start;
   point           p1, p2;
   int             plane_number;
{
   int             count, boolean = FALSE;
   polygon_list   *polygon = NULL;
   obstacle_plane *current = NULL;
   point           p3, p4;
   double          w, x, y, z;

   for (polygon = start; polygon != NULL;
     polygon = polygon->next_polygon) {

      for (count = 1, current = polygon->polygon_start, boolean = FALSE;
        count <= polygon->polygon_start->number_nodes;
        count++, current = current->ccw) {
      if (polygon->plane_number != plane_number) {
         boolean = TRUE;
         break;
      }

      if (p2.x == current->pt.x && p2.y == current->pt.y) {
         boolean = TRUE;
         break;
      }
      }
      if (!boolean) {

      for (count = 1, current = polygon->polygon_start;
           count <= polygon->polygon_start->number_nodes;
           count++, current = current->ccw) {
         p3.x = current->pt.x, p3.y = current->pt.y;
         p4.x = current->ccw->pt.x, p4.y = current->ccw->pt.y;

         if ((p1.x == current->pt.x && p1.y == current->pt.y) || (p1.x
                == current->ccw->pt.x && p1.y == current->ccw->pt.y))
            continue;


         if (point_on_line(current->pt, current->ccw->pt, p1)) {
            continue;     /* skips over p1 which is on the polygon */
         }

         w = sign(function_one(p3.x, p3.y, p1, p2));
         x = sign(function_one(p4.x, p4.y, p1, p2));
         y = sign(function_two(p1.x, p1.y, p3, p4));
         z = sign(function_two(p2.x, p2.y, p3, p4));

         if (w != x && y != z)
            return (1);   /* line intersects polygon */

      }
      current = current->ccw;
      }
   }
```

```
   return (0);
}

/*-------------------------------------------------------------------------
                                REVERSE LIST
-------------------------------------o0o-----------------------------------
This function identifies the shortest path and reverses it so it is from
start to goal vice goal to start order.  The function builds a new path
in doing so returns the path with a pointer labeled reverse path*/

paths          *reversed_list(path_list)
   paths          *path_list;
{
   paths          *reversed_path;
   a_path         *current_path_list, *current_reversed_path, *new_record;

   reversed_path = duplicate_entries(path_list);

   NEW_NODES(reversed_path->last_node);
   current_reversed_path = reversed_path->last_node;
   current_reversed_path->pt = assign_point_values(start_point);
   current_reversed_path->next_node = NULL;

   for (current_path_list = path_list->last_node;
     current_path_list != NULL;
     current_path_list = current_path_list->next_node) {

      NEW_NODES(new_record);
      new_record->pt = assign_point_values(current_path_list->pt);
      new_record->next_node = current_reversed_path->next_node;
      current_reversed_path->next_node = new_record;
   }
   return (reversed_path);
}

/*-------------------------------------------------------------------------
                    DIRECTION HEURISTIC FROM START
-------------------------------------o0o-----------------------------------
This function determines if the new node to be added to the start paths
list is valid.  A node is valid if the direction from the start to the
node is within 90 degrees of the start to goal direction.  The initial pt
can either be the start point or the goal point.  If it is the goal pt
then the goal direction is normalized to goal direction minus 180 degrees
*/

int             direction_heuristic_from_start(expand, pt)
   point          expand, pt;

{
   double         ray_direction, a, goal_dir = goal_direction;

   if(goal.x == expand.x && goal.y == expand.y && goal.z == expand.z)
     goal_dir = norm(goal_dir + PI);

   ray_direction = atan2((pt.y - expand.y), (pt.x - expand.x));
   a = ABS(norm(ray_direction - goal_dir));
   if (a <= HPI)
      return (1);
   else
      return (0);
}


/*-------------------------------------------------------------------------
                    DIRECTION HEURISTIC
-------------------------------------o0o-----------------------------------
This function determines if the new node to be added to a valid path is
in the correct direction.  If the type of tangent to the last node is a
```

```
plus tangent, the the new direction minus the old direction must be less
than or equal to zero.  Vice-a-versa for a minus tangens*/

int             direction_heuristic(current, pt)
   paths          *current;
   point           pt;
{
   double          ray_direction, a, goal_dir = goal_direction;
   a_path         *node = current->last_node;

   if(!current->forward)
     goal_dir = norm(goal_dir + PI);

   ray_direction = atan2((pt.y - node->pt.y),
                (pt.x - node->pt.x));

   a = ray_direction - goal_dir;
   a = norm(a);
   if (ABS(a) > HPI)
      return (0);

   a = norm(ray_direction - current->ray_direction);
   if (ABS(a) > PI / 2)
      return (0);
/*   else
       return (1);*/

   if (current->type_tangent == PLUS) {
    if (norm(ray_direction - current->ray_direction) <= 0)
      return (1);
      else
      return (0);
   }
   else {
      if (norm(ray_direction - current->ray_direction) >= 0)
      return (1);
      else
      return (0);
   }
)

/*--------------------------------------------------------------------
                           NORM
-------------------------------------OOO-------------------------------
This function normalizes an angle between -PI  and PI.*/

double          norm(a)
   double          a;
{
   while ((a > PI) || (a <= -PI)) {
      if (a > PI)
      a = a - DPI;
      else
      a = a + DPI;
   }
   return (a);
}

/*--------------------------------------------------------------------
                        UPDATE CURRENT PATH
-------------------------------------OOO-------------------------------
This  function updates the path information associated with the path
pointed at by the current_path pointer.  After updating the info, the
function returns the start_path pointer back to the calling function.*/

paths          *update_current_path(current_path, /*start_path,*/ tangent,
                                    start_pt, pt, goal, polygon_number)
   paths          *current_path/*, *start_path*/;
```

```
   point           start_pt, pt, goal;
   int             tangent, polygon_number;

{
   current_path->last_node->pt = assign_point_values(pt);
   current_path->last_node->polygon_number = polygon_number;
   current_path->last_node->next_after_node = NULL;
   current_path->cost = distance(start_pt, pt);
   current_path->estimated_cost = distance(goal, pt);
   current_path->total_cost = TOTAL_COST;
   current_path->type_tangent = tangent;
   current_path->ray_direction = atan2((pt.y - start_pt.y),
                        (pt.x - start_pt.x));
   current_path->plane_number = plane_number;
   current_path->able2expand = TRUE;
   current_path->forward = YES;
   head_active_nodes = insert_nodes_into_active_node_list(head_active_nodes, pt,
                          current_path->cost);
   current_path->when_path_found = paths_found++;
   return (current_path);
}

/*-------------------------------------------------------------------------
                          EXPAND PATHS WITH VALUES
-----------------------------------o0o-------------------------------------
This function recalculates the header data of the duplicate record of the
shortest path to include the new point to be added (hence expanded as the
variable name).  It also inserts this expanded path into the link list of
valid paths after the shortest path and returns the shortest path to the
calling function.*/

paths          *expand_paths_with_values(shortest, expanded, tangent_list,
                                head_active, polygon_number, mode)
   paths          *shortest, *expanded;
   a_path         *tangent_list;
   active_nodes   *head_active;
   int             polygon_number, mode;
{
   a_path                      *current  =  shortest->last_node,  *to_polygon  =
tangent_list->next_node;
   int             last_tangent_type = expanded->type_tangent;

   if (shortest->last_node->pt.x == to_polygon->pt.x &&
       shortest->last_node->pt.y == to_polygon->pt.y &&
       shortest->last_node->pt.z == to_polygon->pt.z) {

      expanded->cost += distance(shortest->last_node->pt, tangent_list->pt);
      free(to_polygon);
      tangent_list->next_node = expanded->last_node;
      expanded->last_node->next_after_node = tangent_list;
      expanded->last_node = tangent_list;
   }
   else {
      expanded   =   connect_links_of_path_on_polygon(expanded,   tangent_list,
last_tangent_type);
   }

/*
    expanded->cost   +=   distance(shortest->last_node->pt,
shortest->last_node->next_node->pt);
*/
   if(expanded->forward)
     expanded->estimated_cost = distance(goal,expanded->last_node->pt);
   else
     expanded->estimated_cost = distance(start_point, expanded->last_node->pt);
   expanded->total_cost = expanded->cost + expanded->estimated_cost;
   expanded->ray_direction = atan2((expanded->last_node->pt.y -
                        expanded->last_node->next_node->pt.y),
```

```
                        (expanded->last_node->pt.x -
                         expanded->last_node->next_node->pt.x));
   expanded->able2expand = TRUE;
   expanded->type_tangent = mode;
   head_active = insert_nodes_into_active_node_list(head_active,
                        expanded->last_node->pt, expanded->cost);

/*
expanded = update(expanded, mode);
*/
   expanded->when_path_found = paths_found++;

   expanded->next_path = shortest->next_path;
   expanded->last_node->polygon_number = polygon_number;
   shortest->next_path = expanded;

   return (shortest);
}

/*--------------------------------------------------------------------
                          ADD ON NEW PLANE INTERSECTION
-----------------------------------OOO---------------------------------
This function finds the vertical and perpendicular planes at the point
passed in.  The point is the vertical point to be expanded from.  Once
it finds these planes, the function calls the build_two_d_polygon_list,
which determines the polygons formed by the intersection of the new
perpendicular plane with each polyhedron in the search space.*/

polygon_list   *add_on_new_plane_intersection(start_2d,
                                        vertical_pt_list, goal)
   polygon_list    *start_2d;
   a_path          *vertical_pt_list;
   point            goal;
{
   polygon_list    *last_polygon;
   a_path          *current = vertical_pt_list;

   vertical_n_perpendicular_plane(vertical_pt_list->pt, goal);
   start_2d = build_two_d_polygon_list(start_2d, perpendicular, TRUE);

   return (start_2d);
}

/*--------------------------------------------------------------------
                          POINT ON POLYGON
-----------------------------------OOO---------------------------------
This function determines if a point lies on a line.  The function is used
to skip the polyhedron edge which the vertical point expanded from lies
on.  It is also is used as a heuristic to determine if a new vertical
point lies between the goal point and the current point being expanded
on.*/

int             point_on_line(pt1, pt2, pt3)
   point           pt1, pt2, pt3;/* pt1 and pt2 are end points of
                                  * line, pt3 is point being checked */
{
   double          t1, t2, t3;

   if (pt2.x == pt1.x) {
      if (pt3.x != pt1.x)
      return (0);
      else {
      if ((pt1.y != pt2.y) && (pt1.z != pt2.z)) {
         t2 = (pt3.y - pt1.y) / (pt2.y - pt1.y);
         t3 = (pt3.z - pt1.z) / (pt2.z - pt1.z);
         if (t2 == t3)
            return (1);
         else
```

```
            return (0);
        }
        else {
            if (pt1.y == pt2.y) {
                if (pt3.y == pt1.y)
                return (1);
                else {
                t3 = (pt3.z - pt1.z) / (pt2.z - pt1.z);
                if (0 <= t3 && t3 <= 1)
                    return (1);
                else
                    return (0);
                }
            }
            else {
                if (pt1.z == pt2.z) {
                if (pt1.z != pt3.z)
                    return (0);
                else {
                    t2 = (pt3.y - pt1.y) / (pt2.y - pt1.y);
                    if (0 <= t2 && t2 <= 1)
                   return (1);
                    else
                   return (0);
                }
                }
            }
        }
        }
    }
    else {
        if (pt2.y == pt1.y) {
        if (pt3.y != pt1.y)
            return (0);
        else {
            if (pt1.z != pt2.z) {
                t1 = (pt3.x - pt1.x) / (pt2.x - pt1.x};
                t3 = (pt3.z - pt1.z) / (pt2.z - pt1.z);
                if (t1 == t3}
                return (1);
                else
                return (0);
            }
            else {
                if (pt1.z == pt2.z) {
                if (pt3.z == pt1.z)
                    t1 = (pt3.x - pt1.x} / (pt2.x - pt1.x);
                if (0 <= t1 && t1 <= 1)
                    return (1};
                else
                    return (0};
                }
            }
        }
        }
        else {
        if (pt1.z == pt2.z) {
            if (pt3.z != pt2.z)
                return (0);
        }
        else {
            t1 = (pt3.x - pt1.x) / (pt2.x - pt1.x);
            t2 = (pt3.y - pt1.y) / (pt2.y - pt1.y);
            if (t1 == t2)
                return (1);
            else {
                t1 = (pt3.x - pt1.x) / (pt2.x - pt1.x);
                t2 = (pt3.y - pt1.y} / (pt2.y - pt1.y);
```

```
            t3 = (pt3.z - pt1.z) / (pt2.z - pt1.z);
            if (t1 == t2 == t3)
             return (1);
            else
             return (0);
          }
       }
       }
   }
   return (0);
}


/*-------------------------------------------------------------------
                        CONNECT LINKS OF PATH ON POLYGON
-----------------------------------o0o---------------------------------
This function connects the links of a path on the polygon.  The tangent
function finds the either a plus or minus tangent between polygons.  This
tangent connects the path between the last node of the expanded path and
the tangent node which lies on the same polygon as that last node.*/

paths          *connect_links_of_path_on_polygon(expanded, tangent_list, mode)
   paths          *expanded;
   a_path         *tangent_list;
   int             mode;
{
   a_path         *last_node_in_tangent = tangent_list->next_node;
   polygon_list   *polygon;
   obstacle_plane *node;
   point           extend;

   extend = expanded->last_node->pt;

   last_node_in_tangent->polygon_number = expanded->last_node->polygon_number;

   /* finds correct polygon points lie on */
   for (polygon = start_2d;
     polygon->polygon_number != expanded->last_node->polygon_number;
     polygon = polygon->next_polygon);

   /*
    * finds the correct node in polygon which is equal to last node in
    * tangent list
    */
   for (node = polygon->polygon_start;
     node->pt.x != last_node_in_tangent->pt.x ||
     node->pt.y != last_node_in_tangent->pt.y ||
     node->pt.z != last_node_in_tangent->pt.z;
     node = node->ccw);

   while (LOOPFOREVER) {
      if (mode == PLUS) {
       if (extend.x == node->cw->pt.x && extend.y == node->cw->pt.y && extend.z ==
node->cw->pt.z)
         break;

      else {
         NEW_NODES(last_node_in_tangent->next_node);
         last_node_in_tangent->next_node->next_after_node = last_node_in_tangent;
         last_node_in_tangent = last_node_in_tangent->next_node;
         last_node_in_tangent->pt = assign_point_values(node->ccw->pt);
      }
      node = node->ccw;
      }
      else {
       if (extend.x == node->ccw->pt.x && extend.y == node->ccw->pt.y && extend.z
== node->ccw->pt.z)
         break;
```

```
      else {
         NEW_NODES(last_node_in_tangent->next_node);
         last_node_in_tangent->next_node->next_after_node = last_node_in_tangent;
         last_node_in_tangent = last_node_in_tangent->next_node;
         last_node_in_tangent->pt = assign_point_values(node->cw->pt);
      }
      node = node->cw;
      }
   }
   expanded->cost = calculate_cost(tangent_list, expanded->last_node,
                        expanded->cost, last_node_in_tangent);
   last_node_in_tangent->next_node = expanded->last_node;
   expanded->last_node->next_after_node = last_node_in_tangent;
   expanded->last_node = tangent_list;
   return (expanded);
}

double          calculate_cost(list, last_node_in_path, cost, last_tangent)
   a_path         *list, *last_node_in_path, *last_tangent;
   double          cost;
{
   a_path         *tangent = last_tangent;

   cost += distance(last_node_in_path->pt, last_tangent->pt);
   head_active_nodes = insert_nodes_into_active_node_list(head_active_nodes,
                              last_tangent->pt, cost);


   for (tangent = last_tangent->next_after_node;
     tangent != NULL;
     tangent = tangent->next_after_node) {

      cost += distance(tangent->pt, tangent->next_node->pt);
      head_active_nodes = insert_nodes_into_active_node_list(head_active_nodes,
                           tangent->pt, cost);

   }
   return (cost);
}

/*-----------------------------------o0o---------------------------------
MARK_SHORTEST_LAST_NODE
-------------------------------------------------------------------------
This function marks the last node's polygon number with the polygon's
number in which the node was found.
*/
paths *mark_shortest_path_last_node(shortest, start2)
  paths *shortest;
  polygon_list *start2;
{
  polygon_list *polygon;
  obstacle_plane *node;
  int count;

  for (polygon = start2;
       polygon != NULL;
       polygon = polygon->next_polygon) {

    if (shortest->plane_number != polygon->plane_number)
      continue;
    for(count = 1, node = polygon->polygon_start;
     count <= polygon->polygon_start->number_nodes;
     node = node->ccw, count++){
      if (node->pt.x != shortest->last_node->pt.x ||
        node->pt.y != shortest->last_node->pt.y ||
        node->pt.z != shortest->last_node->pt.z)
     continue;
      else{
```

```
      shortest->last_node->polygon_number = polygon->polygon_number;
      break;
       }
     }
   }
   return(shortest);

}

/*-----------------------------------------------------------------
UPDATE
-------------------------------OOO-------------------------------
This function accepts a path which has been expanded on and updates
the header information accordingly, passing back the path and updated
header info
*/

paths *update(the_path, tan)
   paths *the_path;
   int tan;
{
  the_path->cost += distance(the_path->last_node->pt,
                the_path->last_node->next_node->pt);
  the_path->estimated_cost = distance(the_path->last_node->pt, goal);
  the_path->total_cost = the_path->cost + the_path->estimated_cost;

  the_path->ray_direction = atan2((the_path->last_node->pt.y -
                        the_path->last_node->next_node->pt.y),
                       (the_path->last_node->pt.x -
                        the_path->last_node->next_node->pt.x));
  the_path->able2expand = TRUE;
  the_path->type_tangent = tan;
  head_active_nodes = insert_nodes_into_active_node_list(head_active_nodes,
                       the_path->last_node->pt, the_path->cost);

  the_path->when_path_found = paths_found++;

  the_path->next_path = the_path->next_path;

  return(the_path);

}

int direction_heuristic4tangent(path,list)
  paths *path;
  a_path *list;
{
  double direction, goal_dir = goal_direction, a;
  a_path *node = path->last_node;

  if(!path->forward)
     goal_dir = norm(goal_dir + PI);

  direction = atan2((list->pt.y - list->next_node->pt.y),
               (list->pt.x - list->next_node->pt.x));

  a = direction - goal_dir;
  a = norm(a);
  if (ABS(a) > HPI)
      return (0);

  if(node->pt.x == list->pt.x &&
    node->pt.y == list->pt.y &&
    node->pt.z == list->pt.z){

    a = norm(path->ray_direction - direction);
    if(ABS(a) > HPI)
      return (0);
```

```
    }
    return(1);
}
```

APPENDIX G

This appendix contains the source code to make the simple plots of the world. This source code is DOS dependent and is not included if the defined variable DOS is equal to 1 during compilation.

```
#include "3d_tan.h"
#include "plot.h"

#ifdef DOS

#define ESC             0x1b/* Define the escape key*/
#define SCALEDX         x / 48
#define SCALEDY         y / 34
#define DOT             1               /*used to make a dot with radius 1*/

void data_plot(start_2d)
  polygon_list *start_2d;
{
  int    x, y, z, w;

  Initialize();
  MainWindow( "PLOT OF THE TWO D ENVIRONMENT" );

  StatusLine( "PRESS ANY KEY TO END" );
  x = getmaxx();
  y = getmaxy();
  z = start_point.x * SCALEDX; w = start_point.y * SCALEDY;
  outtextxy( z, w, "X" );
  outtextxy(z+20, w+15, "START");
  z = goal.x * SCALEDX; w = goal.y * SCALEDY;
  outtextxy( z, w, "X" );
  outtextxy(z + 20, w+15, "GOAL");
  draw_boundry();
  draw_polygons(start_2d);

}


void Initialize(void)
{
  int xasp, yasp;         /* Used to read the aspect ratio*/

  if(registerbgidriver(EGAVGA_driver) < 0) /*checks for correct driver*/
    exit(1);
  if(registerbgifont(triplex_font) < 0) /*checks for correct font*/
    exit(1);

  GraphDriver = DETECT;   /* Request auto-detection*/
  initgraph( &GraphDriver, &GraphMode, "");
  ErrorCode = graphresult();/* Read result of initialization*/
  if( ErrorCode != grOk ){/* Error occured during init*/
    printf(" Graphics System Error: %s\n", grapherrormsg( ErrorCode ) );
    exit( 1 );
  }

  getpalette( &palette ); /* Read the palette from board*/
  MaxColors = getmaxcolor() + 1;/* Read maximum number of colors*/

  MaxX = getmaxx();
  MaxY = getmaxy();       /* Read size of screen*/
```

```
  getaspectratio( &xasp, &yasp );/* read the hardware aspect*/
  AspectRatio = (double)xasp / (double)yasp; /* Get correction factor*/

}

 /*                          */
/*    PAUSE: Pause until the user enters a keystroke. If the*/
/*    key is an ESC, then exit program, else simply return.*/
/*                          */

void Pause(void)
{
  static char msg[] = "Pausing.  Esc aborts or press a key...";
  int c;

  StatusLine( msg );       /* Put msg at bottom of screen*/

  c = getch();             /* Read a character from kbd*/

  if( ESC == c ){          /* Does user wish to leave?*/
    closegraph();          /* Change to text mode*/
    exit( 1 );             /* Return to OS */
  }

  if( 0 == c ){            /* Did use hit a non-ASCII key? */
    c = getch();           /* Read scan code for keyboard*/
  }
  StatusLine("Working on next expansion");

  /*cleardevice();*/       /* Clear the screen*/

}


void MainWindow( char *header )
{
  int height;

  cleardevice();           /* Clear graphics screen*/
  setcolor( MaxColors - 1 );/* Set current color to white*/
  setviewport( 0, 0, MaxX, MaxY, 1 );/* Open port to full screen*/

  height = textheight( "H" );           /* Get basic text height        */

  changetextstyle( DEFAULT_FONT, HORIZ_DIR, 1 );
  settextjustify( CENTER_TEXT, TOP_TEXT );
  outtextxy( MaxX/2, 2, header );
  setviewport( 0, height+4, MaxX, MaxY-(height+4), 1 );
  DrawBorder();
  setviewport( 1, height+5, MaxX-1, MaxY-(height+5), 1 );

}

/*                          */
/*    DRAWBORDER: Draw a solid single line around the current */
/*    viewport.             */
/*                          */

void DrawBorder(void)
{
  struct viewporttype vp;

  setcolor( MaxColors - 1 );/* Set current color to white*/

  setlinestyle( SOLID_LINE, 0, NORM_WIDTH );

  getviewsettings( &vp );
  rectangle( 0, 0, vp.right-vp.left, vp.bottom-vp.top );
```

```
}
/*                          */
/*    STATUSLINE: Display a status line at the bottom of the screen.*/
/*                          */

void StatusLine( char *msg )
{
  int height;

  setviewport( 0, 0, MaxX, MaxY, 1 );/* Open port to full screen*/
  setcolor( MaxColors - 1 );/* Set current color to white*/

  changetextstyle( DEFAULT_FONT, HORIZ_DIR, 1 );
  settextjustify( CENTER_TEXT, TOP_TEXT );
  setlinestyle( SOLID_LINE, 0, NORM_WIDTH );
  setfillstyle( EMPTY_FILL, 0 );

  height = textheight( "H" );           /* Detemine current height      */
  bar( 0, MaxY-(height+4), MaxX, MaxY );
  rectangle( 0, MaxY-(height+4), MaxX, MaxY );
  outtextxy( MaxX/2, MaxY-(height+2), msg );
  setviewport( 1, height+5, MaxX-1, MaxY-(height+5), 1 );

}

void changetextstyle(int font, int direction, int charsize)
{
  int ErrorCode;

  graphresult();          /* clear error code*/
  settextstyle(font, direction, charsize);
  ErrorCode = graphresult();/* check result */
  if( ErrorCode != grOk ){/* if error occured*/
    closegraph();
    printf(" Graphics System Error: %s\n", grapherrormsg( ErrorCode ) );
    exit( 1 );
  }
}

void draw_boundry()
{ int x, y;

  x = getmaxx();
  y = getmaxy();
  setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
  setcolor(4);
  line(boundry.x1 * SCALEDX, boundry.y1 * SCALEDY, boundry.x1 * SCALEDX,
       boundry.y2 * SCALEDY);
  line(boundry.x1 * SCALEDX, boundry.y1 * SCALEDY, boundry.x2 * SCALEDX,
       boundry.y1 * SCALEDY);
  line(boundry.x2 * SCALEDX, boundry.y1 * SCALEDY, boundry.x2 * SCALEDX,
       boundry.y2 * SCALEDY);
  line(boundry.x1 * SCALEDX, boundry.y2 * SCALEDY, boundry.x2 * SCALEDX,
       boundry.y2 * SCALEDY);
}



void draw_polygons(start_2d)
  polygon_list *start_2d;
{
  polygon_list *poly;
  obstacle_plane *node;
  int count = 1, x, y;

  x = getmaxx();
  y = getmaxy();
```

```
  setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
  for (poly = start_2d; poly != NULL; poly = poly->next_polygon){
    for (node = poly->polygon_start, count = 1;
         count <= poly->polygon_start->number_nodes;
         node = node->ccw, count++){

         line(node->pt.x * SCALEDX, node->pt.y * SCALEDY,
              node->ccw->pt.x * SCALEDX, node->ccw->pt.y * SCALEDY);
    }
  }
}


void draw_shortest_path(shortest)
  paths *shortest;
{
  paths *path;
  a_path *point;
  int x, y, w, z, style;

  x = getmaxx();
  y = getmaxy();
  style = THICK_WIDTH;
  setcolor(14);
  setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
  for(point = shortest->last_node; point != NULL;
      point = point->next_node){
    if(point->next_node != NULL)
      draw_line_segment(point->pt, point->next_node->pt, YELLOW, style);
      /*(point->pt.x * SCALEDX, point->pt.y * SCALEDY,
            point->next_node->pt.x * SCALEDX, point->next_node->pt.y * SCALEDY);*/
    circle(point->pt.x * SCALEDX, point->pt.y * SCALEDY, DOT);
  }
  Pause();
  cleardevice();
}


void draw_line_segment(pt1, pt2, color, style)
     point pt1, pt2;
     int color, style;
{
  int x, y, w, z;

  x = getmaxx();
  y = getmaxy();
  setcolor(color);
  if(color == RED || style)
    setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
  else
    setlinestyle(DOTTED_LINE, 0, NORM_WIDTH);
   line(pt2.x * SCALEDX, pt2.y * SCALEDY, pt1.x * SCALEDX, pt1.y * SCALEDY);
  z = pt1.x * SCALEDX; w = pt1.y * SCALEDY;
  outtextxy( z, w, "X" );
  z = pt2.x * SCALEDX; w = pt2.y * SCALEDY;
  outtextxy( z, w, "X" );
}

void id_shortest_path_for_expansion(shortest)
  paths *shortest;
{

  int x, y, color, i;
  point p1, p2;
  a_path *node;

  x = getmaxx();
  y = getmaxy();
```

```
  setcolor(BLACK);
  outtextxy(Expansion_lable_x * SCALEDX,
            Expansion_lable_y * SCALEDY, "X" );
  outtextxy(Expansion_lable_x * SCALEDX + 20,
            Expansion_lable_y * SCALEDY + 15, "EXPAND");

  for(i = 1, node = shortest->last_node;
      node->next_node != NULL;
      node = node->next_node, i++){
    p1 = node->pt;
    p2 = node->next_node->pt;

    draw_line_segment(p1, p2, RED, REGULAR);

    if(i == 1){
      Expansion_lable_x = p1.x;
      Expansion_lable_y = p1.y;
      outtextxy( p1.x * SCALEDX, p1.y * SCALEDY, "X" );
      outtextxy(p1.x * SCALEDX + 20, p1.y * SCALEDY + 15, "EXPAND");
    }
  }
  Pause();

}

#endif
```

APPENDIX H

This appendix contains the source code in print.c and has only those functions associated with printing to either a file or the screen. It also contains some print utilities I used in debugging other portions of my code

```
#include "3d_tan.h"

void            print_polyhedron(list)
   obstacle_list  *list;
{
   obstacle_list  *current_obstacle = polyhedron_list;
   polyhedron_obstacle *current_face;
   polyhedron_obstacle_plane *current;
   int             count1, count2, count3;

   for (count1 = 1, current_obstacle = list;
     count1 <= number_polygons;
     count1++,
     current_obstacle = current_obstacle->next_polygon) {
      printf("\nObstacle %d\n", count1);
      for (current_face = current_obstacle->obstacle, count2 = 1;
        current_face != NULL;
        current_face = current_face->next_face, count2++) {
      printf("\nplane face %d\n", count2);
      for (current = current_face->face_nodes, count3 = 1;
           count3 <= current_face->face_nodes->number_nodes;
           count3++, current = current->ccw) {

         printf("\nx = %.3lf, y = %.3lf, z =  %.3lf",
             current->pt.x, current->pt.y, current->pt.z);
      }
      }
   }
}

/*-------------------------------------------------------------------------
                        PRINT RESULTS
-----------------------------------o0o-----------------------------------*/
void            print_results(polyhedron_list, number_polygons,
                              start_point, goal, plane_equation)
   obstacle_list  *polyhedron_list;
   equation        plane_equation;
   point           start_point, goal;
{
   int             count1, count2, count3;
   obstacle_list  *current_obstacle = polyhedron_list;
   polyhedron_obstacle *current_face;
   polyhedron_obstacle_plane *current;
   polyhedron_obstacle_plane *pt1 = NULL, *pt2 = NULL, *pt3 = NULL;

   for (count1 = 1, current_obstacle = polyhedron_list;
     count1 <= number_polygons;
     count1++,
     current_obstacle = current_obstacle->next_polygon) {
      printf("\nObstacle %d\n", count1);

      for (current_face = current_obstacle->obstacle, count2 = 1;
        current_face != NULL;
        current_face = current_face->next_face, count2++) {

      printf("\nplane face %d\n", count2);
```

```
      for (current = current_face->face_nodes, count3 = 1;
           count3 <= current_face->face_nodes->number_nodes;
           count3++, current = current->ccw) {

         printf("\nx = %.3lf, y = %.3lf, z =  %.3lf",
             current->pt.x, current->pt.y, current->pt.z);
      }
      }
   }

   /* check */
   printf("\nstart point is %.3lf, %.3lf, %.3lf\n",
       start_point.x, start_point.y, start_point.z);
   printf("\ngoal is %.3lf, %.3lf, %.3lf\n",
       goal.x, goal.y, goal.z);

   /* check plane equation for first face */
   current_face = polyhedron_list->obstacle;
   for (count2 = 1, current_obstacle = polyhedron_list;
     current_obstacle != NULL;
     current_obstacle = current_obstacle->next_polygon,
     current_face = polyhedron_list->obstacle) {

      for (count1 = 1, current_face = current_obstacle->obstacle;
        current_face != NULL;
        current_face = current_face->next_face, count1++) {

      printf("face %d\n", count1);
      pt1 = current_face->face_nodes;
      pt2 = pt1->ccw;
      pt3 = pt2->ccw;
      printf("\nx = %.3lf, y = %.3lf, z =  %.3lf",
          pt1->pt.x, pt1->pt.y, pt1->pt.z);
      printf("\nx = %.3lf, y = %.3lf, z =  %.3lf",
          pt2->pt.x, pt2->pt.y, pt2->pt.z);
      printf("\nx = %.3lf, y = %.3lf, z =  %.3lf",
          pt3->pt.x, pt3->pt.y, pt3->pt.z);

      plane_equation = find_plane_equation(pt1->pt, pt2->pt,
                                pt3->pt);
      printf("plane equation = <%.3f, %.3f, %.3f, %.3f>\n",
          plane_equation.x, plane_equation.y,
          plane_equation.z, plane_equation.d);
      }
   }
}

/*-------------------------------------------------------------------------
                                PRINT 2D PATH
-----------------------------------o0o-----------------------------------*/

void            print_2d_path(start_two_d)
   polygon_list   *start_two_d;
{
   polygon_list   *current_polygon = start_two_d;
   obstacle_plane *current;
   int             count1, count2;

   for (count1 = 1; current_polygon != NULL;
     current_polygon = current_polygon->next_polygon, count1++) {
      if (current_polygon->polygon_start == NULL)
      continue;

      printf("\nPOLYGON %d\nprint ccw\n", count1);

      for (count2 = 1, current = current_polygon->polygon_start;
        count2 <= current_polygon->polygon_start->number_nodes;
        count2++, current = current->ccw) {
```

```
      printf("x coord = %8.3lf, y coord = %8.3lf, z coord = %8.3lf\n",
          current->pt.x, current->pt.y, current->pt.z);
      }
      printf("print cw\n");
      for (count2 = 1, current = current_polygon->polygon_start;
        count2 <= current_polygon->polygon_start->number_nodes;
        count2++, current = current->cw) {
      printf("x coord = %8.3lf, y coord = %8.3lf, z coord = %8.3lf\n",
          current->pt.x, current->pt.y, current->pt.z);
      }
   }
}

/*-------------------------------------------------------------------------
                              PRINT VERT PERP PATH
-------------------------------------o0o-----------------------------------*/

void            print_vert_perp_plane()
{                   /* all variables are globals */
   printf("\nThe line intersects the polygon at (%.3f, %.3f, %.3f)",
       intersection.x, intersection.y, intersection.z);
   printf("\nThe vertical plane through the start and goal is\n");
   printf("%.3fI + %.3fJ +  %.3fK * %.3f",
       vertical.x, vertical.y, vertical.z, vertical.d);
   printf("\nThe plane through the start and goal perpendicular to the");
   printf(" vertical plane is\n %.3fI + %.3fJ +  %.3fK * %.3f",
       perpendicular.x, perpendicular.y,
       perpendicular.z, perpendicular.d);
}

/*-------------------------------------------------------------------------
                                PRINT EACH PATH
-------------------------------------o0o-----------------------------------*/

void            print_each_path(start_paths, iteration_count)
   paths          *start_paths;
{
   paths          *current_path;
   int             count;

   if (!plot2screen || print2file) {
      if (print2file)
      if(1 == iteration_count)
         fprintf(fpt2, "\n INITIAL PARTIAL PATHS\n");
      else
        fprintf(fpt2, "\n\nPATH FINDING ITERATION COUNT = %d", iteration_count++);
      for (current_path = start_paths, count = 1;
        current_path != NULL;
        count++, current_path = current_path->next_path) {

      if (!plot2screen)
         printf("\n\nPATH %d\n", count);
      if (print2file)
         fprintf(fpt2, "\n\nPATH %d\n", count);

      print_path(current_path);
      }
   }
}

/*-------------------------------------------------------------------------
                              PRINT INTERSECTION
-------------------------------------o0o-----------------------------------*/
void            print_intersection(plane_equation, intersection, pt1, pt2)
   equation        plane_equation;
   point           pt1, pt2, intersection;
{
   printf("\nThe intersection of the line,");
```

```
   printf("\n(%.3f,%.3f, %.3f) to (%.3f,%.3f, %.3f)\nand the plane",
       pt1.x, pt1.y, pt1.z, pt2.x, pt2.y, pt2.z);
   printf("(%.3fI + %.3fJ + %.3fK + %.3f) is (%.3f, %.3f, %.3f)",
       plane_equation.x, plane_equation.y,
       plane_equation.z, plane_equation.d,
       intersection.x, intersection.y, intersection.z);
}

/*-------------------------------------------------------------------------
                      PRINT PATH
-----------------------------------o0o-------------------------------------*/

void            print_path(current_path)
   paths          *current_path;
{
   a_path         *node;

   if (!plot2screen) {
     printf("PATH NUMBER %d  ", current_path->when_path_found);
     if(current_path->able2expand)
       printf("Path able to extend\n");
     else
       printf("Path cannot be extended\n");

     if (current_path->type_tangent == PLUS)
     printf("\nPlus Tangent, direction = %4.3lf\n",
           current_path->ray_direction);
     else
     printf("\nMinus Tangent, direction = %4.3lf\n",
                current_path->ray_direction);
     for (node = current_path->last_node;
        node != NULL;
        node = node->next_node) {

      if (node->next_node != NULL) {
         printf("(%3.2lf, %3.2lf, %3.2lf) to",
              node->pt.x, node->pt.y, node->pt.z);
         printf("\ncost = %4.3lf, est.cost = %4.3lf, total = %4.3lf",
              current_path->cost, current_path->estimated_cost,
              current_path->total_cost);

      }
      else {
         printf("(%3.2lf, %3.2lf, %3.2lf)",
              node->pt.x, node->pt.y, node->pt.z);
         printf("\ncost = %4.3lf, est.cost = %4.3lf, total = %4.3lf",
              current_path->cost, current_path->estimated_cost,
              current_path->total_cost);
      }
     }
   )
   if (print2file)
      print_path_to_file(current_path);

}

void            print_path_to_file(current_path)
   paths          *current_path;

{
   a_path         *node;

     fprintf(fpt2, "PATH NUMBER %d  ", current_path->when_path_found);
     if(current_path->able2expand)
       fprintf(fpt2, "Path able to extend\n");
     else
       fprintf(fpt2, "Path cannot be extended\n");
```

```
   fprintf(fpt2, "cost = %4.3lf, est.cost = %4.3lf, total = %4.3lf\n",
           current_path->cost, current_path->estimated_cost,
           current_path->total_cost);

   if (current_path->type_tangent == PLUS)
      fprintf(fpt2, "\nPlus Tangent, direction = %4.3lf\n",
              current_path->ray_direction);
   else
      fprintf(fpt2, "\nMinus Tangent, direction = %4.3lf\n",
              current_path->ray_direction);
   for (node = current_path->last_node;
     node != NULL;
     node = node->next_node) {
      if (node->next_node != NULL) {
      fprintf(fpt2, "(%3.2lf, %3.2lf, %3.2lf) to\n",
           node->pt.x, node->pt.y, node->pt.z);
      }
      else {
      fprintf(fpt2, "(%3.2lf, %3.2lf, %3.2lf)\n",
           node->pt.x, node->pt.y, node->pt.z);
      }
   }
}

/*-------------------------------------------------------------------------
                     PRINT PATH SHORTEST
-------------------------------------o0o-------------------------------------*/
void            print_path_shortest(shortest)
   paths          *shortest;
{
   a_path         *current;

   if (!plot2screen)
      printf("\nSHORTEST PATH");
   if (print2file)
      fprintf(fpt2, "SHORTEST PATH\n");

   for (current = shortest->last_node;
     current != NULL;
     current = current->next_node) {
      if (!plot2screen) {
      printf("\n(%3.2lf, %3.2lf, %3.2lf)",
          current->pt.x, current->pt.y, current->pt.z);
      if (current->next_node != NULL)
         printf(" to");
      }
      if (print2file) {
      fprintf(fpt2, "\n(%3.2lf, %3.2lf, %3.2lf)",
           current->pt.x, current->pt.y, current->pt.z);
      if (current->next_node != NULL)
         fprintf(fpt2, " to");
      }
   }
   if (!plot2screen)
      printf("\nThe total length of the shortest path is %4.3lf\n",
          shortest->total_cost);
   if (print2file)
      fprintf(fpt2, "\nThe total length of the shortest path is %4.3lf\n",
           shortest->total_cost);

}
```

APPENDIX I

This function contains the source code found in tangent.c and contains only those functions which are associated with tangents.

```
#include "3d_tan.h"
/*-----------------------------------------o0o---------------------------------
TANGENT
-----------------------------------------o0o---------------------------------
This function finds the tangent if one exist between two polygons.  If any
obstacles lie between the two polygons, then the function returns NULL.  If a
tangent line is found then the points are placed in a list in euclidean order and
returned.
*/
a_path         *tangent(shortest, polygon, second_polygon_start, mode,
               start_2d, type_path, mode2)
   paths          *shortest;
   polygon_list  *polygon;
   obstacle_plane *second_polygon_start;
   int             mode;
   polygon_list   *start_2d;
   int            type_path;
   int            mode2;
{
   a_path         *tangent_path_list = NULL;
   point          to_polygon_pt, from_polygon_pt;
   int             i, points_found = 0, tan;

   if(type_path)
      tan = shortest->type_tangent;
   else
      tan = shortest->polygon4vert_PTR->type_tangent;

   if(type_path) mode2 = shortest->type_tangent;

   if (mode2 /*shortest->type_tangent*/ == MINUS) {
      if (mode == MINUS) {
      for(i = 1;
          i <= second_polygon_start->number_nodes &&
            i <= polygon->polygon_start->number_nodes;
          i++){

        to_polygon_pt = minus_tangent(second_polygon_start,
                          shortest->last_node->pt);
        from_polygon_pt = plus_tangent(polygon->polygon_start, to_polygon_pt);
        if (!cross_polygon_node_on(start_2d, to_polygon_pt, from_polygon_pt,
                          polygon->plane_number)){
           points_found = TRUE;

           break;
        }
      }
   }
   else {
      for(i = 1;
          i <= second_polygon_start->number_nodes &&
            i <= polygon->polygon_start->number_nodes;
          i++){
        to_polygon_pt = plus_tangent(second_polygon_start,
                          shortest->last_node->pt);

        from_polygon_pt = plus_tangent(polygon->polygon_start, to_polygon_pt);
```

```
        if (!cross_polygon_node_on(start_2d, to_polygon_pt, from_polygon_pt,
                        polygon->plane_number)){
           points_found = TRUE;

           break;
        }
      }
    }
   }
   else {
    if (mode == MINUS) {
      for(i = 1;
          i <= second_polygon_start->number_nodes &&
          i <= polygon->polygon_start->number_nodes;
          i++){
        to_polygon_pt = minus_tangent(second_polygon_start,
                                      shortest->last_node->pt);
        from_polygon_pt = minus_tangent(polygon->polygon_start, to_polygon_pt);
        if (!cross_polygon_node_on(start_2d, to_polygon_pt, from_polygon_pt,
                        polygon->plane_number)){
           points_found = TRUE;

           break;
        }
      }
    }
    else {
      for(i = 1;
          i <= second_polygon_start->number_nodes &&
            i <= polygon->polygon_start->number_nodes;
          i++){
        to_polygon_pt = plus_tangent(second_polygon_start,
                                     shortest->last_node->pt);
        from_polygon_pt = minus_tangent(polygon->polygon_start, to_polygon_pt);
        if (!cross_polygon_node_on(start_2d, to_polygon_pt, from_polygon_pt,
                        polygon->plane_number)){
           points_found = TRUE;

           break;
        }
      }
     }
   }
   if(points_found)
     tangent_path_list = make_tangent_list(to_polygon_pt, from_polygon_pt);
    return (tangent_path_list);
}


a_path          *make_tangent_list(to_point, from_point)
    point           to_point, from_point;

{
    a_path          *list;

    NEW_NODES(list);
    NEW_NODES(list->next_node);
    list->next_node->next_node = NULL;
    list->next_node->next_after_node = list;
    list->next_after_node = NULL;

    list->pt = assign_point_values(to_point);
    list->next_node->pt = assign_point_values(from_point);
    return (list);
}
```

```
/*-------------------------------------------------------------------------
                                PLUS TANGENT
-------------------------------------o0o-----------------------------------
This functions finds and returns the node which with the inputted point
forms the plus common tangent.  It calls the the order function.  Sign is
a macro and is defined in 3d_tan.h and returns an integer corresponding
to the value returned by order.  The function returns the point found to
the calling function*/

point           plus_tangent(start, p1)
   obstacle_plane *start;
   point           p1;
{
   obstacle_plane *next = start->ccw, *prev = start->cw, *current = start;
   int             count;
   point           pt;
   double          dir1, dir2;

   for (count = 1; count <= start->number_nodes; count++) {

     if (current->pt.x == p1.x && current->pt.y == p1.y) {
        current = current->ccw;
        break;
      }

      if (sign(order(p1, current, next)) == 1) {
         current = next;
         next = next->ccw;
      }
      else {
         if (sign(order(p1, current, prev)) == 1) {
           current = prev;
           prev = prev->cw;
         }
         else
           break;
      }
    }
    if(atan2(p1.y - current->pt.y, p1.x - current->pt.x) ==
       atan2(p1.y - current->ccw->pt.y, p1.x - current->ccw->pt.x)){
      if(distance(p1, current->pt) < distance(p1, current->ccw->pt))
        return(current->pt);
      else
        return(current->ccw->pt);
    }
    if(atan2(p1.y - current->pt.y, p1.x - current->pt.x) ==
       atan2(p1.y - current->cw->pt.y, p1.x - current->cw->pt.x)){
      if(distance(p1, current->pt) < distance(p1, current->cw->pt))
        return(current->pt);
      else
        return(current->cw->pt);
    }
    return (current->pt);
}
```

```
/*-------------------------------------------------------------------
                        MINUS TANGENT
-----------------------------------o0o---------------------------------
finds and returns the node in the polygon which, with the inputted point
forms the minus common tangent.  It calls the the order function.  Sign
is a macro and is defined in 3d_tan.h and returns an integer corresponding
to the value returned by order.  The function returns the point found to
the calling function*/

point           minus_tangent(start, p1)
   obstacle_plane *start;
   point           p1;
{
   obstacle_plane *next = start->cw, *prev = start->ccw, *current = start;
   int             count;
   point           pt;

   for (count = 1; count <= start->number_nodes; count++) {

      if (current->pt.x == p1.x && current->pt.y == p1.y) {
         current = current->cw;
         break;
      }

      if (sign(order(p1, current, next)) == -1) {
         prev = current;
         current = next;
         next = next->cw;
      }
      else {
         if (sign(order(p1, current, prev)) == -1) {
           next = current;
           current = prev;
           prev = prev->ccw;
         }
         else
           break;
      }
    }
    if(atan2(p1.y - current->pt.y, p1.x - current->pt.x) ==
       atan2(p1.y - current->ccw->pt.y, p1.x - current->ccw->pt.x)){
      if(distance(p1, current->pt) < distance(p1, current->ccw->pt))
        return(current->pt);
      else
        return(current->ccw->pt);
    }
    if(atan2(p1.y - current->pt.y, p1.x - current->pt.x) ==
       atan2(p1.y - current->cw->pt.y, p1.x - current->cw->pt.x)){
      if(distance(p1, current->pt) < distance(p1, current->cw->pt))
        return(current->pt);
      else
        return(current->cw->pt);
     }
     return (current->pt);
}
```

APPENDIX J

This appendix contains the source code of vertical.c. This code corresponds to those functions used to find the vertical path over a polyhedron.

```
#include "3d_tan.h"
#include "plot.h"

#define LOOP

/*-------------------------------------------------------------------------
                    FIND VERTICAL NODES
-------------------------------------000-------------------------------
This function finds the appropriate vertical points formed by the inter-
section of the vertical plane and each polyhedron.  The function cycles
through the polyhedron list checking each polyhedron and finding all
intersection points for that polyhedron.  It then determines the two
highest points and checks checks if the intersection points are between
the expanded point and the goal.  If the points are, the function calls
new_vertical_new_list, which places one of the two or both of the points
into a vertical list (link list).  The function then checks the next
polyhedron, finding the appropriate intersection points and calls the
new_vertical_new_list which replaces the points in the vertical list if
they satisfy the requirments of the heuristics in that function.  It
recieves the vertical list from new_vertical_new_list and passes it to
the calling function upon completion.*/

vertical_path  *find_vertical_nodes(expand_pt, end_pt)
   point           expand_pt, end_pt;   /* expand_pt is the point from
                                         * which the vertical list is
                                         * formed, end_point is the
                                         * ending point that is used
                                         * to form the vertical list */

{
   point           pt1, pt2, temp;
   obstacle_list  *current_polyhedron;
   polyhedron_obstacle *face;
   polyhedron_obstacle_plane *current_node;
   int             count, boolean = FALSE, two_points = FALSE;
   vertical_path *vertical_list = NULL;

   vertical_n_perpendicular_plane(expand_pt, end_pt);

   for (current_polyhedron = polyhedron_list;
         current_polyhedron != NULL;
         current_polyhedron = current_polyhedron->next_polygon,
         two_points = FALSE, boolean = FALSE) {

      for (face = current_polyhedron->obstacle;
            face != NULL; face = face->next_face) {

       for (current_node = face->face_nodes, count = 1;
            count <= face->face_nodes->number_nodes;
            current_node = current_node->ccw, count++) {
          if (two_points == FALSE) {
             if (boolean == FALSE) {
                pt1 = intersection_point(current_node->pt,
                                 current_node->ccw->pt, vertical);
                if (pt1.x != NO_INTERSECTION) {
                  boolean = TRUE;
                  continue;
```

```
                }
            }
            else {
                pt2 = intersection_point(current_node->pt,
                            current_node->ccw->pt, vertical);
                if (pt2.x != NO_INTERSECTION) {
                  two_points = TRUE;
                  if (pt1.z > pt2.z) {
                     temp = swap(pt1);
                     pt1 = swap(pt2);
                     pt2 = swap(temp);
                  }
                }
            }
          }
          else {
            temp = intersection_point(current_node->pt,
                          current_node->ccw->pt, vertical);
            if (temp.x != NO_INTERSECTION) {
                if (temp.x == pt1.x && temp.y == pt1.y && temp.z == pt1.z) {
                  if (temp.x == pt2.x && temp.y == pt2.y && temp.z == pt2.z) {
                     continue;
                  }
                }
                if (temp.z < pt1.z) {
                  pt2 = swap(pt1);
                  pt1 = swap(temp);
                }
                else {
                  if (temp.z < pt2.z)
                     pt2 = swap(temp);
                }
            }
          }
        }
      }
       if (pt1.x != NO_INTERSECTION && boundry_check(boundry,pt1) &&
          boundry_check(boundry, pt2) &&
           (pt1.x != end_pt.x || pt1.y != end_pt.y) && (pt1.x != expand_pt.x ||
          pt1.y != expand_pt.y) &&
           (pt2.x != expand_pt.x || pt2.y != expand_pt.y)) { /*the or checks are
                            to ensure the expand or end point are not included as
                            in the case of a face edge*/
         if (vertical_node_direction_heuristic(end_pt, expand_pt, pt1) &&
             vertical_node_direction_heuristic(end_pt, expand_pt, pt2))

           vertical_list = vertical_node_list_from_pt(vertical_list,pt1,pt2,
                                expand_pt, current_polyhedron->polyhedron_number);

       }
    }
    return (vertical_list);
}


point           swap(pt)
    point           pt;
{
    point           pt1;

    pt1.x = pt.x;
    pt1.y = pt.y;
    pt1.z = pt.z;
    return (pt1);
}
```

```
/*-------------------------------------------------------------------
                    NEW VERTICAL NODE LIST FROM POINT
-------------------------------------------------------------------
This function uses the two vertical points passed to it from find_verti-
cal_nodes, which represent the vertical nodes from a polyhedron, and
places them in eucledian distance order in a vertical_list.  It returns
to the calling function this vertical_list.
*/
vertical_path *vertical_node_list_from_pt(vertical_list,ptl,pt2, pt, poly_num)
   vertical_path *vertical_list;
   point          ptl, pt2, pt;
   int            poly_num;
{
    double          ptl_distance, pt2_distance, pt_dist;
    a_path          *current, *previous, *temp;

   ptl_distance = distance(ptl, pt);
   pt2_distance = distance(pt2, pt);
   NEW_NODES(temp);
   temp->next_node = NULL;

   if(ptl_distance == pt2_distance){
     temp->pt = assign_point_values(ptl);
     temp->polygon_number = poly_num;
   }

   if(ptl_distance != pt2_distance){
     NEW_NODES(temp->next_node);
     temp->next_node->next_node = NULL;
     if(ptl_distance > pt2_distance){
       temp->pt = assign_point_values(ptl);
       temp->next_node->pt = assign_point_values(pt2);
     }
     else{
       temp->pt = assign_point_values(pt2);
       temp->next_node->pt = assign_point_values(ptl);
     }
     temp->polygon_number = poly_num;
     temp->next_node->polygon_number = poly_num;
   }

   if(vertical_list == NULL){
     NEW_VERTICAL(vertical_list);
     vertical_list->next_node = temp;
   }
   else{
     for(current = vertical_list->next_node, previous = NULL;
      current != NULL || temp == NULL;
      previous = current->next_node,
       current = current->next_node->next_node){

      pt_dist = distance(pt, current->pt);
      if(ptl_distance > pt_dist ||  pt2_distance > pt_dist){
     temp->next_node->next_node = current;
     if(previous == NULL){
       vertical_list->next_node = temp;
       break;
     }
     else
       previous->next_node = temp;
      }
      if(current->next_node == NULL){  /*considers only one node in list
                          and temp nodes come after it*/
     current->next_node = temp;
     break;
      }
     }
     if (temp != NULL && previous != NULL)
```

```
        previous->next_node = temp;
    }
    vertical_list->next_node->next_after_node = NULL;
    return(vertical_list);
}


/*-------------------------------------------------------------------
                          INSERT VERTICAL LIST
-------------------------------------OOO-------------------------------
This function takes the vertical list and inserts the list as a valid
path in the start_paths list.
*/
paths   *insert_verticle_list(start_paths, current_path, vertical_list, tan4vert,
                    end_pt, polygon4vert)

     paths          *start_paths, *current_path;
     vertical_path  *vertical_list;
     int             tan4vert;
     point end_pt;
{
    paths *temp;
    a_path *current;
    double cost = 0;

    NEW_PATH(temp);
    temp->last_node = vertical_list->next_node;
    current = temp->last_node;
    current->next_after_node = NULL;
    temp->last_node->next_after_node = NULL;
    free(vertical_list);

    /*connect  the double link list and calculate the distance from the
      first node to the last node*/
    for( ; LOOP ; current = current->next_node){
      if(current->next_node != NULL){
        current->next_node->next_after_node = current;
        cost += distance(current->pt, current->next_node->pt);
      }
      else
        break;
    }
    temp->polygon4vert = polygon4vert;
    temp->polygon4vert_PTR = NULL;
    temp->cost = cost;
    temp->estimated_cost = distance(temp->last_node->pt, goal);
    temp->total_cost = cost + temp->estimated_cost;
    temp->type_tangent = VERTICAL;
    temp->ray_direction = atan2((temp->last_node->pt.y -
                                 temp->last_node->next_node->pt.y),
                                 (temp->last_node->pt.x -
                                 temp->last_node->next_node->pt.x));
    temp->plane_number = 0;
    temp->when_path_found = paths_found++;
    temp->able2expand = TRUE;
    temp->polygon4vert_PTR = input_data4polyhed4vert(temp->polygon4vert_PTR,
                                                     end_pt, tan4vert);

    temp->next_path = current_path->next_path;
    if(start_paths == NULL)
      start_paths = temp;
    else{
      temp->next_path = current_path->next_path;
      current_path->next_path = temp;
    }
```

```
#ifdef DOS
   if (plot2screen){
     for(current = temp->last_node;
      current->next_node != NULL;
      current = current->next_node)
           draw_line_segment(current->next_node->pt, current->pt, YELLOW,
                                  REGULAR);
   }
#endif

   return (start_paths);
}



/*-------------------------------------------------------------------------
                        VERTICAL NODE DIRECTION HEURISTIC
-------------------------------------OOO---------------------------------
This function determines if a point lies in the same xy direction from a
start point as the direction of the line from the start_pt to the goal.
The start_pt is not the same as the global start_point but just a local
variable and could any valid point in the valid possible path list
*/
int             vertical_node_direction_heuristic(goal, expand_pt, pt)
   point           goal, expand_pt, pt;
{
   double          x, y, a;

   if(goal.x == expand_pt.x || pt.x == expand_pt.x)
       expand_pt.x += .0000001;
   if(goal.x == pt.x)
     goal = increase_goal_x(goal, expand_pt);

   x = atan2(goal.y - expand_pt.y, goal.x - expand_pt.x);
   y = atan2(goal.y - pt.y, goal.x - pt.x);
   a = x - y;

   if (ABS(a) <= QPI)
      return (1);
   else
     return (0);
}

/*-------------------------------------------------------------------------
                                  INCREASE GOAL X
-------------------------------------OOO---------------------------------
This function is called if its determined that the increase and pt2 points
have the same value for the x coordinate.  if so then it adds of subtracts
.000001 to the x component goal so that the new distance from the two
points is greater than the original distance.  If not done, atan2 is
undetermined since x - x == 0
*/
point increase_goal_x(increase, pt2)
   point increase, pt2;
{
  double orginal_distance;

  orginal_distance = distance(increase, pt2);

  increase.x += .0000001;
  if(distance(increase, pt2) < orginal_distance)
    increase.x -= .0000002;

  return(increase);

}
```

```
/*-------------------------------------------------------------
                              UP AND OVER PATHS
-------------------------------OOO-------------------------------
This function calls two other functions.  The first finds the verti-
cal nodes of a possible path and places them in a list labled verti-
cal_list.  The second function takes the vertical_list, places it
in the possible paths list and updates the header list of the path.
*/

paths   *up_and_over(start_paths, current_path, exp_pt, end_pt, beginpt, mode,
polygon4vert)
     paths          *start_paths, *current_path;
     point          exp_pt, end_pt, beginpt;
     int            mode, polygon4vert;
{
   vertical_path  *vertical_list = NULL;
   a_path *current;

   vertical_list = find_vertical_nodes(exp_pt, end_pt);
   if (vertical_list != NULL){

     if(beginpt.x == exp_pt.x && beginpt.y == exp_pt.y &&
         beginpt.z == exp_pt.z){
       for(current = vertical_list->next_node;
            current->next_node != NULL;
            current = current->next_node);

       NEW_NODES(current->next_node);

       current->next_node->next_node = NULL;
       current->next_node->pt = assign_point_values(exp_pt);
     }
     vertical_list = trim_not_needed_nodes(vertical_list);
     start_paths = insert_verticle_list(start_paths, current_path,
                          vertical_list, mode, end_pt,
                          polygon4vert);
   }
   if(VERTICAL != mode){}
   return(start_paths);
}

/*-------------------------------------------------------------
                            CONTINUE VERTICAL PATH
-------------------------------OOO-------------------------------
This function finds the one node/tangent which can extend the verti-
cal path by one node on the perpendicular plane formed by the verti-
cal node and the goal.  There is only one node to consider since it
falls on the line segment of the polyhedron used to find the verti-
node
*/
paths *continue_vertical_path(shortest, start2d)
  paths *shortest;
  polygon_list *start2d;
{
  polygon_list *polygon = start2d;
  point pt1, pt2;
  a_path *temp;
  int tan,found = FALSE;
  double ray_direction, difference;

  for (; polygon != NULL; polygon = polygon->next_polygon){
    if(polygon->plane_number != shortest->plane_number)
      continue;
    if(polygon->polyhedron_number != shortest->polygon4vert)
      continue;
    else
      break;
  }
```

```
    if(l == shortest->polygon4vert_PTR->type_tangent)
      pt1 = plus_tangent(polygon->polygon_start, shortest->last_node->pt);
    else
      pt1 = minus_tangent(polygon->polygon_start, shortest->last_node->pt);

    ray_direction = atan2(pt1.y - shortest->last_node->pt.y,
                          pt1.x - shortest->last_node->pt.x);

    /*used le-3 instead of 0 because of the precision of DOS machine. No
     points used evaluated true and the point which should have evalueated
     true, had a difference of le-10*/

    difference = shortest->ray_direction - ray_direction;

    /*find the up_around and up paths before continuing on*/

    if(ABS(difference) < le-3){
      NEW_NODES(temp);
      temp->pt = assign_point_values(pt1);
      temp->polygon_number = polygon->polygon_number;
      temp->next_node = shortest->last_node;
      shortest->last_node = temp;
      if(shortest->polygon4vert_PTR->type_tangent) tan = PLUS;
      else tan = MINUS;
      shortest = update(shortest, tan);
      found = TRUE;
    }
    else{
      shortest = recalculate_vertical_path(pt1, shortest,
                           (shortest->polygon4vert_PTR->type_tangent    ?PLUS:
MINUS));
      shortest->last_node->polygon_number = polygon->polygon_number;
      found = TRUE;
    }

  if (!found)
    shortest->able2expand = NO;

#ifdef DOS
  else{
    if (plot2screen)
      draw_line_segment(shortest->last_node->next_node->pt,
               shortest->last_node->pt, YELLOW, REGULAR);

  }
#endif

  return(shortest);
}


/*-----------------------------------------------------------------
                              UP AND OVER FROM GOAL
-------------------------------OOO-------------------------------
This function finds all the vertical partial paths from the goal
extending to the start.  This working backwards from the goal allows
me to find the valid paths which go around one or more obstacles and
over the rest of them.
*/
paths *up_and_over_from_goal(paths *start_paths, paths *current_path,
                     point goal, polygon_list *start2d)
 /* paths *start_paths, *current_path;
  point goal;
  polygon_list *start2d;
  */
{
  polygon_list *current_polygon;
  point pt1, pt2;
```

```
  int poly4pt;

  for (current_polygon = start_2d;
     current_polygon != NULL;
     current_polygon = current_polygon->next_polygon) {

    pt1 = plus_tangent(current_polygon->polygon_start, goal);
    pt2 = minus_tangent(current_polygon->polygon_start, goal);

    if (cross_polygon(start_2d, goal, pt1) &&
     direction_heuristic_from_start(goal, pt1) &&
     boundry_check(boundry, pt1)) {

      start_paths = up_and_over(start_paths, current_path, goal,
                  pt1, goal, PLUS, current_polygon->polyhedron_number);
      if (current_path == NULL) current_path = start_paths;
      else current_path = current_path->next_path;
      current_path->forward = NO;
    }
    if (cross_polygon(start_2d, goal, pt2) &&
     direction_heuristic_from_start(goal, pt2) &&
     boundry_check(boundry, pt2)) {

      start_paths = up_and_over(start_paths, current_path, goal,
                  pt2, goal, MINUS);
      if (current_path == NULL) current_path = start_paths;
      else current_path = current_path->next_path;
      current_path->forward = NO;
      current_path->polygon4vert = current_polygon->polyhedron_number;
    }
  }
  return(start_paths);
}

paths *recalculate_vertical_path(pt, path, tan)
  point pt;
  paths *path;
  int tan;
{
  vertical_path *vertical;
  a_path *node = path->last_node,
      *current, *previous;
  double cost = 0;
  int count, number_nodes;

  /*find the last node (or beginning pt of path)*/
  for( ;node->next_node != NULL; node = node->next_node);

  /*disconnect the last node from the path*/
  node->next_after_node->next_node = NULL;
  node->next_after_node = NULL;

  /*delete the path nodes of the shortest_path
   while counting number of nodes*/
  for(current = path->last_node, previous = current, number_nodes = 0;
      current != NULL;
      previous = current, current= current->next_node, number_nodes++)
    free(previous);

  free(previous);

  /*find the new vertical list*/
  vertical = find_vertical_nodes(node->pt, pt);

  /*finds the end of the vertical list then working backwards up the
    list, places the same number of nodes in the list*/
  for(current = vertical->next_node; current->next_node != NULL;
      current = current->next_node)
```

```
      current->next_node->next_after_node = current;

  for(count = 1; count < number_nodes;
      count++, current= current->next_after_node);
  vertical->next_node = current;
  /*frees up the unused portion of the list*/
  for(current = current->next_after_node; current != NULL;
      current = current->next_after_node)

    free(current);

  path->last_node = vertical->next_node;
  free(vertical);
  path = insert_pt_into_list(path, pt);
  path->last_node->next_after_node = NULL;

  for(current = path->last_node;
      current->next_node != NULL;
      current = current->next_node){
    current->next_node->next_after_node = current;
    cost += distance(current->pt, current->next_node->pt);
  }
  current->next_node = node;
  node->next_after_node = current;
  cost += distance(current->pt, node->pt);
  if(path->forward == YES)
    path->estimated_cost = distance(node->pt, goal);
  else
    path->estimated_cost = distance(node->pt, start_point);

  path->total_cost = cost + path->estimated_cost;
  path->cost = cost;
  path->ray_direction = atan2(path->last_node->pt.y -
                              path->last_node->next_node->pt.y,
                                   path->last_node->pt.x -
                              path->last_node->next_node->pt.x);
  path->type_tangent = tan;

  return(path);
}

paths *insert_pt_into_list(path, pt)
  paths *path;
  point pt;
{
  a_path *temp = NEW_NODES(temp);
  temp->pt = assign_point_values(pt);
  temp->next_after_node = NULL;
  temp->next_node = path->last_node;
  path->last_node = temp;

  return(path);
}

polyhed4 *input_data4polyhed4vert(ptr, end_pt, mode)
  polyhed4 *ptr;
  point end_pt;
  int mode;
{
  if(NULL == ptr)
    ptr = (polyhed4 *) malloc(sizeof(polyhed4));

  ptr->x = end_pt.x;
  ptr->y = end_pt.y;
  ptr->type_tangent = mode;

  return(ptr);
}
```

```
vertical_path *trim_not_needed_nodes(list)
  vertical_path *list;
{
  a_path *node1, *node2, *node3, *node4;

  for(node1 = list->next_node;
      NULL != node1;
      node1 = node1->next_node){

    if(NULL == node1->next_node->next_node) break;

    for(node2 = node1->next_node->next_node;
     NULL != node2;
     node2 = node2->next_node){

      if(intersect_polyhedron(node1->pt, node2->pt)) continue;
      else{
          for(node3 = node1->next_node, node4 = node3;
              node2 != node3;
              node4 = node3, node3 = node3->next_node,free(node4));

          node1->next_node = node2;
          node2->next_after_node = node1;
      }
    }
  }
  return(list);
}
```

APPENDIX K

This appendix contains the source code associated with visibile.c.

```
#include "3d_tan.h"
#include "plot.h"

#define LOOPFOREVER 1


/*-----------------------------------o0o---------------------------------
                                VISIBILITY
-----------------------------------o0o---------------------------------
This function determines whether or no the last point in the partial path is
visible to the goal or not.  The function considers whether a point on the
backside of a polygon is visible to the goal by first finding the tangents to
the goal with the polygon and traces the path around to the expansion point.
A point is valid in this case if no obstacles lie between the goal and the
tangent point
*/
a_path *visibility(expand_point, goal_point, start, two_d_list)
  point  expand_point, goal_point, start;
  polygon_list *two_d_list;
{
  polygon_list *polygon;
  point from_polygon_pt, goal;
  a_path *temp, *temp1, *goal_list;
  obstacle_plane *node;
  int tangent = 1;

  goal = determine_end_point(start, goal_point);
  if(!intersect_polyhedron(expand_point, goal)){
    NEW_NODES(goal_list);
    goal_list->next_node = NULL;
    goal_list->pt = assign_point_values(goal);
    return(goal_list);
  }
  if(shortest_path->type_tangent == VERTICAL)
    return(NULL);
  else{
    /*find polygon which last node of shortest path lies on*/
    for(polygon = two_d_list;
     polygon->polygon_number != shortest_path->last_node->polygon_number ||
     polygon->plane_number != shortest_path->plane_number;
     polygon = polygon->next_polygon);

    while(tangent <= 2){
      if(1 == tangent)
     /*finds the plus tangent from the goal to the polygon last node of
       shortest path lies on*/
     from_polygon_pt = plus_tangent(polygon->polygon_start, goal);
      else
     /*check the minus tangent*/
     from_polygon_pt = minus_tangent(polygon->polygon_start, goal);

      /*checks for intersection, if there is return a zero*/
      if(intersect_polyhedron(from_polygon_pt, goal))
     return(NULL);
      else{           /*two points are visible*/
     if(temp == NULL){
       NEW_NODES(temp);
```

```
  temp->next_node = NULL;
  temp->next_after_node = NULL;
}
temp->pt = assign_point_values(from_polygon_pt);
temp1 = temp;

/*finds the node on the polygon which the from_polygon_pt lies on*/
for (node = polygon->polygon_start;
  node->pt.x != from_polygon_pt.x ||
  node->pt.y != from_polygon_pt.y ||
  node->pt.z != from_polygon_pt.z;
  node = node->ccw);

while (LOOPFOREVER) {
  if(1 == tangent){
    /*checks to see if the next counter clockwise node is the
      expansion node*/
    if(node->ccw->pt.x == shortest_path->last_node->pt.x &&
       node->ccw->pt.y == shortest_path->last_node->pt.y &&
       node->ccw->pt.z == shortest_path->last_node->pt.z)
      break;
    else{
      if(temp1->next_node == NULL){
     NEW_NODES(temp1->next_node);
     temp1->next_node->next_after_node = temp1;
     temp1->next_node->next_node = NULL;
      }
      temp1 = temp1->next_node;
      temp1->pt = assign_point_values(node->ccw->pt);
    }
    node = node->ccw;
  }
  else{ /*tangent == 2*/
    /*checks to see if the next counter clockwise node is the
      expansion node*/
    if(node->cw->pt.x == shortest_path->last_node->pt.x &&
       node->cw->pt.y == shortest_path->last_node->pt.y &&
       node->cw->pt.z == shortest_path->last_node->pt.z){
      if(temp1->next_node != NULL){
     for(temp1 = temp1->next_node,
         temp1->next_after_node->next_node = NULL,
         temp1->next_after_node = NULL,
         temp1->next_node;
         temp1->next_node != NULL;
         temp1 = temp1->next_node)
       free(temp1->next_after_node);
     if(temp1->next_after_node != NULL)
       free(temp1->next_after_node);
     free(temp1);
      }
      for(temp1 = temp; temp->next_node != NULL; temp = temp->next_node);
    break;
    }
    else{
      if(temp1->next_node == NULL){
     NEW_NODES(temp1->next_node);
     temp1->next_node->next_after_node = temp1;
     temp1->next_node->next_node = NULL;
      }
      temp1 = temp1->next_node;
      temp1->pt = assign_point_values(node->cw->pt);
    }
    node = node->cw;
  }
}
if (direction_heuristic(shortest_path, temp1->pt)){
  goal_list = temp;
  return(goal_list);
```

```
      }
       }
       tangent += 1;
     }
   }
   for(temp1 = temp->next_node; temp1->next_node != NULL; temp1 = temp1->next_node)
     free(temp1->next_after_node);
   free(temp->next_after_node);
   free(temp);
   return(NULL);
}


point determine_end_point(start, goal)
   point start, goal;
{
     if(shortest_path->forward) return(goal);
     else return(start);
}
```

APPENDIX L

This appendix contains the two header files (3d_tan.h and plot.h).

```
#include <math.h>
#include <stdio.h>

/*-------------------------------------------------------------------------
                        3D TANGENT HEADER FILE (3D_TAN.H)
-------------------------------------o0o-----------------------------------*/

#define  READONLY       "r"
#define  WRITEONLY      "w"
#define  TRUE           1
#define  FALSE          0
#define  NO             0
#define  YES            1
#define  NO_INTERSECTION   999999.9990
#define  sqr(x)             (x * x)
#define  F(X,Y,Z)       ((pl_eq.x*temp_pt.x) + (pl_eq.z*temp_pt.z))/pl_eq.y
#define  VERTICAL       2
#define  NONE           2
#define  PLUS           1
#define  MINUS          0
#define  GOAL_PT        1
#define  NOT_GOAL_PT    0
#define sign(a)         (((a) < 0) ? -1 : (a) > 0 ? 1 : 0)
#define ABS(x)          (((x) < 0) ? -x : (x) > 0 ? x : 0)
#define TOTAL_COST      current_path->cost + current_path->estimated_cost
#define FOREVER         1
#define QPI             0.785398230785398
#define HPI             1.5707963257949
#define PI              3.141593123141593
#define DPI             6.283186246283186
#define NEW_OBSTACLE(x)   ((x = (obstacle *) malloc
                            (sizeof (obstacle)))==NULL?exit(1):1)
#define NEW_PLANE(x)      ((x = (obstacle_plane  *) malloc
                            (sizeof (obstacle_plane )))==NULL?exit(1):1)
#define NEW_NODE(x)       ((x = (obstacle_list *) malloc
                            (sizeof(obstacle_list))) == NULL?exit(1):1)
#define NEW_POLYHEDRON(x) ((x = (polyhedron_obstacle *) malloc
                            (sizeof(polyhedron_obstacle))) == NULL?exit(1):1)
#define NEW_POLY_PLANE(x) ((x = (polyhedron_obstacle_plane *) malloc
                           (sizeof(polyhedron_obstacle_plane))) == NULL?exit(1):1)
#define NEW_VERTICAL(x)   ((x = (vertical_path *) malloc
                            (sizeof(vertical_path))) == NULL?exit(1):1)
#define NEW_GOAL(x)       ((x = (goals *) malloc
                            (sizeof(goals))) == NULL?exit(1): 1)
#define NEW_PATH(x)       ((x = (paths *) malloc
                            (sizeof(paths))) == NULL?exit(1): 1)
#define NEW_NODES(x)      ((x = (a_path *) malloc
                            (sizeof(a_path))) == NULL?exit(1):1)
#define NEW_POLYGON(x)    ((x = (polygon_list *) malloc
                            (sizeof(polygon_list))) == NULL?exit(1):1)
#define NEW_ACTIVE_NODE(x) ((x = (active_nodes *) malloc
                            (sizeof(active_nodes))) == NULL?exit(1):1)
#define POINT_VALUES   &x_coord, &y_coord, &z_coord
```

```
#ifdef MAIN
#  define EXTERN
#  define INIT(Value) = Value
#else
#  define EXTERN extern
#  define INIT(Value)
#endif


typedef struct polyhedron_list {
   struct polyhedron *obstacle;
   int             number_faces;
   int    polyhedron_number;
   struct polyhedron_list *next_polygon;
}              obstacle_list;

EXTERN obstacle_list *polyhedron_list INIT(NULL);
EXTERN obstacle_list *intersected_polyhedron INIT(NULL);

typedef struct polyhedron {
   struct polyhedron_node *face_nodes;
   int             number_nodes;
   struct polyhedron *next_face;
}              polyhedron_obstacle, polyhedron_face;

typedef struct posture {
   double          x;
   double          y;
   double          z;
   double          phi;
   double          t;
   double          psi;
}              POSTURE;

typedef struct x_y_z {
   double          x;
   double          y;
   double          z;
}              point;

typedef struct polyhedron_node {
   struct x_y_z    pt;
   int             number_nodes;
   struct polyhedron_node *cw;
   struct polyhedron_node *ccw;
}   polyhedron_obstacle_plane, polyhedron_nodes, obstacle_plane;

EXTERN point    start_point, goal, intersection;

typedef struct plane_equation {
   double          x;
   double          y;
   double          z;
   double          d;
}              equation;

EXTERN equation vertical, perpendicular;

typedef struct two_d_polygon_list {
   obstacle_plane *polygon_start;
   struct two_d_polygon_list *next_polygon;
   int             plane_number;
   int             polygon_number;
   int             polyhedron_number;
}              polygon_list;

EXTERN polygon_list *start_2d INIT(NULL);
```

```
/*typedef struct polygon_node {
   struct x_y_z    pt;
   int             number_nodes;
   struct polygon_node *ccw;
   struct polygon_node *cw;
}              obstacle_plane;*/

typedef struct list_of_active_nodes {
   point           pt;
   double          distance;
   struct list_of_active_nodes *next_active_node;
   int             when_path_found;
}              active_nodes;

EXTERN active_nodes *head_active_nodes INIT(NULL);

typedef struct list_of_paths {
   struct list_of_paths *next_path;
   struct possible_paths *last_node;
   double          cost;
   double          estimated_cost;
   double          total_cost;
   double          ray_direction;
   int             type_tangent;
   int             plane_number;
   int             when_path_found;
   int             able2expand;
   int             forward;
   int             polygon4vert;
   struct polyhed  *polygon4vert_PTR;
}              paths;

EXTERN path_and_node_list *head_path_node_list INIT(NULL);
EXTERN paths   *start_path INIT(NULL);
EXTERN paths   *shortest_path INIT(NULL);

typedef struct possible_paths {
   point           pt;
   struct possible_paths *next_node;
   struct possible_paths *next_after_node;
   int  polygon_number;
}              a_path, goals;

typedef struct vertical{
   double            distance;
   struct possible_paths *next_node;
}   vertical_path;

EXTERN goals   *goal_path INIT(NULL);

typedef struct polyhed{
    double x,
        y;
    int    type_tangent;
}polyhed4;

typedef struct boundry_limits {
   double          x1;
   double          x2;
   double          y1;
   double          y2;
   double          z1;
   double          z2;
}              boundries;

EXTERN boundries boundry;
```

```
EXTERN char     file[11];
EXTERN char    *filename INIT(file);
EXTERN int paths_found INIT(0);
EXTERN int plane_number INIT(0);
EXTERN int print2file INIT(0);
EXTERN int horizontal INIT(0);
EXTERN int plot2screen INIT(0);
EXTERN int on_same_polygon INIT(0);
EXTERN int      number_polygons;
EXTERN int number_paths_found INIT(0);
EXTERN double goal_direction INIT(0);
EXTERN FILE    *fpt2 INIT(NULL);

/* function declarations */

extern active_nodes *insert_nodes_into_active_node_list();

extern a_path *free_tangent_list();
extern a_path *make_tangent_list();
extern a_path *tangent();
extern a_path *visibility();

extern boundries assign_boundry_points();

extern double   function_one();
extern double   function_two();
extern double   norm();
extern double   order();

extern int      boundry_check();
extern int      check_active_node_list();
extern int      check_for_node_on_polygon();
extern int      cross_polygon();
extern int      cross_polygon_node_on();
extern int      count_polygon_nodes();
extern int      decision();
extern int      intersect_polyhedron();
extern int      lines_intersect();
extern int      point_on_line();
extern int      vertical_node_direction_heuristic();
extern int      direction_heuristic_from_start();
extern int      direction_heuristic();

extern equation cross_product();
extern equation find_plane_equation();

extern double calculate_cost();
extern double   distance();

char           *malloc();

extern obstacle_list *create_obstacle_list();

extern obstacle_plane *create_list();

extern paths   *add_start_point_to_path();
extern paths   *create_path_n_nodes();
extern paths   *connect_links_of_path_on_polygon();
extern paths   *connect_shortest_paths();
extern paths   *continue_vertical_path();
extern paths   *delete_before_final();
extern paths   *duplicate_entries();
extern paths   *duplicate_paths();
extern paths   *expand_paths_with_values();
extern paths   *extend_path();
extern paths   *find_tangents_from_inner_node();
extern paths   *goal_reached_update();
extern paths   *insert_pt_into_list();
```

```
extern paths   *insert_verticle_list();
extern paths   *insert_vertical_list_into_shortest_path();
extern paths   *mark_old_path_with_node_unexpandable();
extern paths   *mark_shortest_path_last_node();
extern paths   *up_and_over();
extern paths   *up_and_over_from_goal();
extern paths   *up_around_up();
extern paths   *recalculate_vertical_path();
extern paths   *remove_shortest_path();
extern paths   *remove_paths();
extern paths   *reversed_list();
extern paths   *start_finding_paths();
extern paths   *update_current_path();
extern paths   *update();

extern point    assign_point_values();
extern point    determine_end_point();
extern point    find_shortest_path();
extern point    increase_goal_x();
extern point    input_point();
extern point    intersection_point();
extern point    plus_tangent();
extern point    minus_tangent();
extern point    find_point_on_obstacle_plane_face();
extern point    swap();

extern polygon_list *add_on_new_plane_intersection();
extern polygon_list *build_polygon();
extern polygon_list *build_two_d_polygon_list();
extern polygon_list *connect_links();
extern polygon_list *free_polygon_list();
extern polygon_list *make_list();
extern polygon_list *remove_duplicate_nodes();
extern polygon_list *remove_last_polygon();
extern polygon_list *remove_planes();

extern polyhedron_face *find_polyhedron_face();

extern polyhedron_obstacle *create_face();
extern polyhedron_obstacle_plane *create_plane();

extern polyhed4   *input_data4polyhed4vert();

extern void     read_comment();
extern void     find_all_paths();
extern void     free_memory();
extern void     input_start_goal();
extern void     inside_boundries();
extern void     load_goal_path();
extern void     vertical_n_perpendicular_plane();
extern void     print_intersection();
extern void     print_results();
extern void     print_2d_path();
extern void     print_each_paths();
extern void     print_path();
extern void     print_path_shortest();
extern void     print_polyhedron();
extern void     print_path_to_file();

extern vertical_path *find_vertical_nodes();
extern vertical_path  *new_vertical_new_list();
extern vertical_path *trim_not_needed_nodes();
extern vertical_path *vertical_node_list_from_pt();
```

```
--------------------------------------------------------------------------------
                                    PLOT.H
--------------------------------------------------------------------------------

#include <graphics.h>
#include <dos.h>

#define DOS             1
#define ESC             0x1b/* Define the escape key*/
#define SCALEDX         x / 48
#define SCALEDY         y / 34
#define DOT             1               /*used to make a dot with radius 1*/
#define RED             12
#define YELLOW          14
#define REGULAR         0

#ifdef MAIN
#  define EXTERN
#  define INIT(Value) = Value
#else
#  define EXTERN extern
#  define INIT(Value)
#endif

EXTERN int    GraphDriver INIT(0);/* The Graphics device driver*/
EXTERN int    GraphMode INIT(0);/* The Graphics mode value*/
EXTERN int    MaxX, MaxY INIT(0);/* The maximum resolution of the screen */
EXTERN int    MaxColors INIT(0);/* The maximum # of colors available*/
EXTERN int    ErrorCode INIT(0);/* Reports any graphics errors*/
EXTERN double AspectRatio INIT(0.0);/* Aspect ratio of a pixel on the screen*/
EXTERN struct palettetype palette;/* Used to read palette info*/
EXTERN int    Expansion_lable_x INIT(0);/* Used to label the expansion point*/
EXTERN int    Expansion_lable_y INIT(0);

extern void data_plot();
extern void Initialize();
extern void Pause();
extern void MainWindow();
extern void DrawBorder();
extern void StatusLine();
extern void changetextstyle();
extern void draw_boundry();
extern void draw_polygons();
extern void draw_shortest_path();

extern void draw_line_segment();
extern void id_shortest_path_for_expansion();
```

APPENDIX M

This first example is for 1 polyhedron and corresponds to the first example in Chapter IV.

```
 INITIAL PARTIAL PATHS

PATH 1
PATH NUMBER 0  Path able to extend
cost = 11.589, est.cost = 31.192, total = 42.781

Plus Tangent, direction = 0.661
(10.00, 20.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 11.014, est.cost = 31.811, total = 42.825

Minus Tangent, direction = -0.588
(10.00, 7.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 13.236, est.cost = 28.263, total = 41.499

Minus Tangent, direction = 0.051
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)


THE GOAL HAS BEEN REACHED

PATH FINDING ITERATION COUNT = 2

PATH 1
PATH NUMBER 0  Path able to extend
cost = 11.589, est.cost = 31.192, total = 42.781

Plus Tangent, direction = 0.661
(10.00, 20.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 11.014, est.cost = 31.811, total = 42.825

Minus Tangent, direction = -0.588
(10.00, 7.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 13.236, est.cost = 28.263, total = 41.499

Minus Tangent, direction = 0.051
```

```
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
SHORTEST PATH

(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
The total length of the shortest path is 41.499


PATH FINDING ITERATION COUNT = 3

PATH 1
PATH NUMBER 0  Path able to extend
cost = 11.589, est.cost = 31.192, total = 42.781

Plus Tangent, direction = 0.661
(10.00, 20.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 11.014, est.cost = 31.811, total = 42.825

Minus Tangent, direction = -0.588
(10.00, 7.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 41.499, est.cost = 0.000, total = 41.499

Minus Tangent, direction = 0.051
(40.00, 15.00, 14.00) to
(40.00, 15.00, 14.00) to
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
SHORTEST PATH

(40.00, 15.00, 14.00) to
(40.00, 15.00, 14.00) to
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
The total length of the shortest path is 41.499


This example is the listing generated by the algorithm for the second example for
a one polyhedral world in Chapter IV.

 INITIAL PARTIAL PATHS

PATH 1
PATH NUMBER 0  Path able to extend
cost = 11.589, est.cost = 31.192, total = 42.781

Plus Tangent, direction = 0.661
(10.00, 20.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 11.014, est.cost = 31.811, total = 42.825
```

```
Minus Tangent, direction = -0.588
(10.00, 7.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 13.236, est.cost = 28.263, total = 41.499

Minus Tangent, direction = 0.051
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)


THE GOAL HAS BEEN REACHED

PATH FINDING ITERATION COUNT = 2

PATH 1
PATH NUMBER 0  Path able to extend
cost = 11.589, est.cost = 31.192, total = 42.781

Plus Tangent, direction = 0.661
(10.00, 20.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 11.014, est.cost = 31.811, total = 42.825

Minus Tangent, direction = -0.588
(10.00, 7.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 13.236, est.cost = 28.263, total = 41.499

Minus Tangent, direction = 0.051
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
SHORTEST PATH

(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
The total length of the shortest path is 41.499


PATH FINDING ITERATION COUNT = 3

PATH 1
PATH NUMBER 0  Path able to extend
cost = 11.589, est.cost = 31.192, total = 42.781

Plus Tangent, direction = 0.661
(10.00, 20.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 11.014, est.cost = 31.811, total = 42.825
```

```
Minus Tangent, direction = -0.588
(10.00, 7.00, 7.08) to
(1.00, 13.00, 5.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 41.499, est.cost = 0.000, total = 41.499

Minus Tangent, direction = 0.051
(40.00, 15.00, 14.00) to
(40.00, 15.00, 14.00) to
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
SHORTEST PATH

(40.00, 15.00, 14.00) to
(40.00, 15.00, 14.00) to
(14.00, 13.67, 3.00) to
(10.00, 13.46, 3.00) to
(1.00, 13.00, 5.00)
The total length of the shortest path is 41.499
```

This next example corresponds to the first example used in a two polyhedral world in Chapter IV.

```
 INITIAL PARTIAL PATHS


PATH 1
PATH NUMBER 0  Path able to extend
cost = 10.296, est.cost = 30.414, total = 40.709

Plus Tangent, direction = 0.507
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 12.042, est.cost = 31.048, total = 43.090

Minus Tangent, direction = -0.727
(10.00, 7.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 18.545, est.cost = 28.438, total = 46.983

Minus Tangent, direction = 0.257
(14.00, 18.42, 3.00) to
(10.00, 17.37, 3.00) to
(1.00, 15.00, 14.00)


PATH 4
PATH NUMBER 3  Path able to extend
cost = 19.049, est.cost = 28.757, total = 47.806

Minus Tangent, direction = -0.399
(14.00, 9.53, 3.00) to
(10.00, 11.21, 3.00) to
(1.00, 15.00, 14.00)
```

```
PATH 5
PATH NUMBER 4  Path able to extend
cost = 28.213, est.cost = 19.416, total = 47.629

Minus Tangent, direction = 0.000
(24.00, 15.00, 3.00) to
(20.00, 15.00, 3.00) to
(14.00, 15.00, 3.00) to
(10.00, 15.00, 3.00) to
(1.00, 15.00, 14.00)


PATH 6
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 7
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879

Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
SHORTEST PATH

(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)
The total length of the shortest path is 40.709


THE GOAL HAS BEEN REACHED

PATH FINDING ITERATION COUNT = 2

PATH 1
PATH NUMBER 0  Path cannot be extended
cost = 10.296, est.cost = 30.414, total = 40.709

Plus Tangent, direction = 0.507
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 2
PATH NUMBER 10  Path able to extend
cost = 28.613, est.cost = 21.541, total = 50.154

Minus Tangent, direction = -1.138
(20.00, 7.00, 14.00) to
(14.00, 20.00, 14.00) to
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 3
PATH NUMBER 8  Path able to extend
cost = 20.296, est.cost = 20.616, total = 40.911

Plus Tangent, direction = 0.000
(20.00, 20.00, 14.00) to
(14.00, 20.00, 14.00) to
```

```
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 4
PATH NUMBER 1  Path able to extend
cost = 12.042, est.cost = 31.048, total = 43.090

Minus Tangent, direction = -0.727
(10.00, 7.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 5
PATH NUMBER 2  Path able to extend
cost = 18.545, est.cost = 28.438, total = 46.983

Minus Tangent, direction = 0.257
(14.00, 18.42, 3.00) to
(10.00, 17.37, 3.00) to
(1.00, 15.00, 14.00)


PATH 6
PATH NUMBER 3  Path able to extend
cost = 19.049, est.cost = 28.757, total = 47.806

Minus Tangent, direction = -0.399
(14.00, 9.53, 3.00) to
(10.00, 11.21, 3.00) to
(1.00, 15.00, 14.00)


PATH 7
PATH NUMBER 4  Path able to extend
cost = 28.213, est.cost = 19.416, total = 47.629

Minus Tangent, direction = 0.000
(24.00, 15.00, 3.00) to
(20.00, 15.00, 3.00) to
(14.00, 15.00, 3.00) to
(10.00, 15.00, 3.00) to
(1.00, 15.00, 14.00)


PATH 8
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 9
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879

Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
SHORTEST PATH

(20.00, 20.00, 14.00) to
(14.00, 20.00, 14.00) to
(10.00, 20.00, 14.00) to
```

```
(1.00, 15.00, 14.00)
The total length of the shortest path is 40.911


PATH FINDING ITERATION COUNT = 3

PATH 1
PATH NUMBER 0  Path cannot be extended
cost = 10.296, est.cost = 30.414, total = 40.709

Plus Tangent, direction = 0.507
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 2
PATH NUMBER 10  Path able to extend
cost = 28.613, est.cost = 21.541, total = 50.154

Minus Tangent, direction = -1.138
(20.00, 7.00, 14.00) to
(14.00, 20.00, 14.00) to
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 3
PATH NUMBER 8  Path able to extend
cost = 41.059, est.cost = 0.000, total = 41.059

Minus Tangent, direction = 0.000
(40.00, 15.00, 14.00) to
(24.00, 20.00, 14.00) to
(20.00, 20.00, 14.00) to
(14.00, 20.00, 14.00) to
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 4
PATH NUMBER 1  Path able to extend
cost = 12.042, est.cost = 31.048, total = 43.090

Minus Tangent, direction = -0.727
(10.00, 7.00, 14.00) to
(1.00, 15.00, 14.00)


PATH 5
PATH NUMBER 2  Path able to extend
cost = 18.545, est.cost = 28.438, total = 46.983

Minus Tangent, direction = 0.257
(14.00, 18.42, 3.00) to
(10.00, 17.37, 3.00) to
(1.00, 15.00, 14.00)


PATH 6
PATH NUMBER 3  Path able to extend
cost = 19.049, est.cost = 28.757, total = 47.806

Minus Tangent, direction = -0.399
(14.00, 9.53, 3.00) to
(10.00, 11.21, 3.00) to
(1.00, 15.00, 14.00)
```

```
PATH 7
PATH NUMBER 4  Path able to extend
cost = 28.213, est.cost = 19.416, total = 47.629

Minus Tangent, direction = 0.000
(24.00, 15.00, 3.00) to
(20.00, 15.00, 3.00) to
(14.00, 15.00, 3.00) to
(10.00, 15.00, 3.00) to
(1.00, 15.00, 14.00)


PATH 8
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 9
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879

Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
SHORTEST PATH

(40.00, 15.00, 14.00) to
(24.00, 20.00, 14.00) to
(20.00, 20.00, 14.00) to
(14.00, 20.00, 14.00) to
(10.00, 20.00, 14.00) to
(1.00, 15.00, 14.00)
The total length of the shortest path is 41.059
```

This final example is the listing generated by the algorithm for the second example for a two polyhedral world in Chapter IV.

```
 INITIAL PARTIAL PATHS

PATH 1
PATH NUMBER 0  Path able to extend
cost = 7.642, est.cost = 31.072, total = 38.715

Plus Tangent, direction = 1.166
(10.00, 20.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 6.738, est.cost = 31.694, total = 38.432

Minus Tangent, direction = -1.107
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 3
PATH NUMBER 2  Path able to extend
cost = 9.797, est.cost = 28.287, total = 38.084

Minus Tangent, direction = 0.494
```

```
(14.00, 16.77, 3.00) to
(10.00, 14.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 4
PATH NUMBER 3  Path able to extend
cost = 9.594, est.cost = 28.712, total = 38.305

Minus Tangent, direction = -0.432
(14.00, 9.77, 3.00) to
(10.00, 11.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 5
PATH NUMBER 4  Path able to extend
cost = 19.029, est.cost = 19.441, total = 38.470

Minus Tangent, direction = 0.061
(24.00, 14.03, 3.00) to
(20.00, 13.79, 3.00) to
(14.00, 13.42, 3.00) to
(10.00, 13.18, 3.00) to
(7.00, 13.00, 7.00)


PATH 6
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 7
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879

Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
SHORTEST PATH

(14.00, 16.77, 3.00) to
(10.00, 14.62, 3.00) to
(7.00, 13.00, 7.00)
The total length of the shortest path is 38.084


PATH FINDING ITERATION COUNT = 2

PATH 1
PATH NUMBER 0  Path able to extend
cost = 7.642, est.cost = 31.072, total = 38.715

Plus Tangent, direction = 1.166
(10.00, 20.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 6.738, est.cost = 31.694, total = 38.432
```

```
Minus Tangent, direction = -1.107
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 3
PATH NUMBER 7  Path able to extend
cost = 17.069, est.cost = 22.284, total = 39.354

Plus Tangent, direction = 0.494
(20.00, 20.00, 5.54) to
(14.00, 16.77, 3.00) to
(10.00, 14.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 4
PATH NUMBER 3  Path able to extend
cost = 9.594, est.cost = 28.712, total = 38.305

Minus Tangent, direction = -0.432
(14.00, 9.77, 3.00) to
(10.00, 11.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 5
PATH NUMBER 4  Path able to extend
cost = 19.029, est.cost = 19.441, total = 38.470

Minus Tangent, direction = 0.061
(24.00, 14.03, 3.00) to
(20.00, 13.79, 3.00) to
(14.00, 13.42, 3.00) to
(10.00, 13.18, 3.00) to
(7.00, 13.00, 7.00)


PATH 6
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 7
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879

Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
SHORTEST PATH

(14.00, 9.77, 3.00) to
(10.00, 11.62, 3.00) to
(7.00, 13.00, 7.00)
The total length of the shortest path is 38.305


PATH FINDING ITERATION COUNT = 3

PATH 1
PATH NUMBER 0  Path able to extend
```

```
cost = 7.642, est.cost = 31.072, total = 38.715

Plus Tangent, direction = 1.166
(10.00, 20.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 2
PATH NUMBER 1  Path able to extend
cost = 6.738, est.cost = 31.694, total = 38.432

Minus Tangent, direction = -1.107
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 3
PATH NUMBER 7  Path able to extend
cost = 17.069, est.cost = 22.284, total = 39.354

Plus Tangent, direction = 0.494
(20.00, 20.00, 5.54) to
(14.00, 16.77, 3.00) to
(10.00, 14.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 4
PATH NUMBER 8  Path able to extend
cost = 16.673, est.cost = 23.143, total = 39.816

Minus Tangent, direction = -0.432
(20.00, 7.00, 5.54) to
(14.00, 9.77, 3.00) to
(10.00, 11.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 5
PATH NUMBER 4  Path able to extend
cost = 19.029, est.cost = 19.441, total = 38.470

Minus Tangent, direction = 0.061
(24.00, 14.03, 3.00) to
(20.00, 13.79, 3.00) to
(14.00, 13.42, 3.00) to
(10.00, 13.18, 3.00) to
(7.00, 13.00, 7.00)


PATH 6
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 7
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879

Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
```

```
SHORTEST PATH

(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)
The total length of the shortest path is 38.432


THE GOAL HAS BEEN REACHED

PATH FINDING ITERATION COUNT = 5

PATH 1
PATH NUMBER 0  Path able to extend
cost = 7.642, est.cost = 31.072, total = 38.715

Plus Tangent, direction = 1.166
(10.00, 20.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 2
PATH NUMBER 1  Path cannot be extended
cost = 6.738, est.cost = 31.694, total = 38.432

Minus Tangent, direction = -1.107
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 3
PATH NUMBER 12  Path able to extend
cost = 16.961, est.cost = 21.954, total = 38.915

Minus Tangent, direction = 0.000
(20.00, 7.00, 9.76) to
(14.00, 7.00, 8.48) to
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 4
PATH NUMBER 10  Path able to extend
cost = 25.202, est.cost = 21.048, total = 46.249

Plus Tangent, direction = 1.138
(20.00, 20.00, 9.76) to
(14.00, 7.00, 8.48) to
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 5
PATH NUMBER 7  Path able to extend
cost = 17.069, est.cost = 22.284, total = 39.354

Plus Tangent, direction = 0.494
(20.00, 20.00, 5.54) to
(14.00, 16.77, 3.00) to
(10.00, 14.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 6
PATH NUMBER 8  Path able to extend
cost = 16.673, est.cost = 23.143, total = 39.816

Minus Tangent, direction = -0.432
(20.00, 7.00, 5.54) to
```

```
(14.00, 9.77, 3.00) to
(10.00, 11.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 7
PATH NUMBER 4  Path able to extend
cost = 19.029, est.cost = 19.441, total = 38.470

Minus Tangent, direction = 0.061
(24.00, 14.03, 3.00) to
(20.00, 13.79, 3.00) to
(14.00, 13.42, 3.00) to
(10.00, 13.18, 3.00) to
(7.00, 13.00, 7.00)


PATH 8
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 9
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879

Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
SHORTEST PATH

(24.00, 14.03, 3.00) to
(20.00, 13.79, 3.00) to
(14.00, 13.42, 3.00) to
(10.00, 13.18, 3.00) to
(7.00, 13.00, 7.00)
The total length of the shortest path is 38.470


PATH FINDING ITERATION COUNT = 6

PATH 1
PATH NUMBER 0  Path able to extend
cost = 7.642, est.cost = 31.072, total = 38.715

Plus Tangent, direction = 1.166
(10.00, 20.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 2
PATH NUMBER 1  Path cannot be extended
cost = 6.738, est.cost = 31.694, total = 38.432

Minus Tangent, direction = -1.107
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 3
PATH NUMBER 12  Path able to extend
cost = 16.961, est.cost = 21.954, total = 38.915
```

```
Minus Tangent, direction = 0.000
(20.00, 7.00, 9.76) to
(14.00, 7.00, 8.48) to
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 4
PATH NUMBER 10  Path able to extend
cost = 25.202, est.cost = 21.048, total = 46.249

Plus Tangent, direction = 1.138
(20.00, 20.00, 9.76) to
(14.00, 7.00, 8.48) to
(10.00, 7.00, 7.64) to
(7.00, 13.00, 7.00)


PATH 5
PATH NUMBER 7  Path able to extend
cost = 17.069, est.cost = 22.284, total = 39.354

Plus Tangent, direction = 0.494
(20.00, 20.00, 5.54) to
(14.00, 16.77, 3.00) to
(10.00, 14.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 6
PATH NUMBER 8  Path able to extend
cost = 16.673, est.cost = 23.143, total = 39.816

Minus Tangent, direction = -0.432
(20.00, 7.00, 5.54) to
(14.00, 9.77, 3.00) to
(10.00, 11.62, 3.00) to
(7.00, 13.00, 7.00)


PATH 7
PATH NUMBER 4  Path able to extend
cost = 38.470, est.cost = 0.000, total = 38.470

Minus Tangent, direction = 0.061
(40.00, 15.00, 14.00) to
(40.00, 15.00, 14.00) to
(24.00, 14.03, 3.00) to
(20.00, 13.79, 3.00) to
(14.00, 13.42, 3.00) to
(10.00, 13.18, 3.00) to
(7.00, 13.00, 7.00)


PATH 8
PATH NUMBER 5  Path able to extend
cost = 24.216, est.cost = 23.640, total = 47.856

Minus Tangent, direction = -2.843
(20.00, 8.85, 3.00) to
(24.00, 10.08, 3.00) to
(40.00, 15.00, 14.00)


PATH 9
PATH NUMBER 6  Path able to extend
cost = 23.732, est.cost = 23.147, total = 46.879
```

```
Minus Tangent, direction = 2.952
(20.00, 18.85, 3.00) to
(24.00, 18.08, 3.00) to
(40.00, 15.00, 14.00)
SHORTEST PATH

(40.00, 15.00, 14.00) to
(40.00, 15.00, 14.00) to
(24.00, 14.03, 3.00) to
(20.00, 13.79, 3.00) to
(14.00, 13.42, 3.00) to
(10.00, 13.18, 3.00) to
(7.00, 13.00, 7.00)
The total length of the shortest path is 38.470
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduates School Monterey, CA 93943-5002	2
3.	Dr. Yutaka Kanayama Code CS/Ka Naval Postgraduates School Monterey, CA 93943	2
4.	CDR Gary Hughes Code CS/Hu Naval Postgraduates School Monterey, CA 93943	1
5.	Dr. Robert McGhee Code CS/Mz Naval Postgraduates School Monterey, CA 93943	1
6.	Dr. Yuh-jeng Lee Code CS/Le Naval Postgraduates School Monterey, CA 93943	1
7.	Dr. Neil Rowe Code CS/Rp Naval Postgraduates School Monterey, CA 93943	1
8.	Dr. Sehung Kwak Code CS/Kw Naval Postgraduates School Monterey, CA 93943	1
9.	Dr. A. J. Healey Code ME/Hy Naval Postgraduates School Monterey, CA 93943	1

10. CPT Tymothy W. Caddell
322 Lynnhaven Drive
Hampton, VA 23666

1

Thesis
C154 Caddell
c.1 Three-dimensional path
planning for the NPS II
AUV.

Thesis
C154 Caddell
c.1 Three-dimensional path
planning for the NPS II
AUV.

DUDLEY KNOX LIBRARY
3 2768 00033071 6